目錄

PyTorch 中文文档

本项目由awfssv, ycszen, KeithYin, kophy, swordspoet, dyl745001196, koshinryuu, tfygg, weigp, ZijunDeng, yichuan9527等PyTorch爱好者发起，并已获得PyTorch官方授权。我们目的是建立PyTorch的中文文档，并力所能及地提供更多的帮助和建议。

本项目网址为pytorch-cn，文档翻译QQ群：628478868

如果你在使用pytorch和pytorch-cn的过程中有任何问题，欢迎在issue中讨论，可能你的问题也是别人的问题。

翻译进度

现在正在进行审阅任务（大家赶紧认领吧～） 第一个名字代表翻译人，第二个代表审阅人

Notes

- [x] Autograd mechanics (*ycszen*)(DL-ljw)
- [x] CUDA semantics (*ycszen*)
- [x] Extending PyTorch (*KeithYin*)
- [x] Multiprocessing best practices (*ycszen*)
- [x] Serialization semantics (*ycszen*)

Package Reference

- [x] torch(*koshinryuu*)(飞彦)
- [x] torch.Tensor(*weigp*)(飞彦)
- [x] torch.Storage(*kophy*)
- [ ] **torch.nn**
  - [x] Parameters(*KeithYin*)
  - [x] Containers(*KeithYin*)
  - [x] Convolution Layers(*yichuan9527*)
  - [x] Pooling Layers(*yichuan9527*)
  - [x] Non-linear Activations(*swordspoet*)
  - [x] Normalization layers(*XavierLin*)
  - [x] Recurrent layers(*KeithYin*)(Mosout)
  - [x] Linear layers( )(Mosout)
  - [x] Dropout layers( )(Mosout)
  - [x] Sparse layers(Mosout)

- [x] Distance functions
- [x] Loss functions(*KeithYin*)(DL-ljw)
- [x] Vision layers(*KeithYin*)
- [x] Multi-GPU layers(*KeithYin*)
- [x] Utilities(*KeithYin*)
- [x] torch.nn.functional
  - [x] Convolution functions(*ycszen*)(铁血丹心)
  - [x] Pooling functions(*ycszen*)(铁血丹心)
  - [x] Non-linear activations functions(*ycszen*)
  - [x] Normalization functions(*ycszen*)
  - [x] Linear functions(*dyl745001196*)
  - [x] Dropout functions(*dyl745001196*)
  - [x] Distance functions(*dyl745001196*)
  - [x] Loss functions(*tfygg*)(DL-ljw)
  - [x] Vision functions(*KeithYin*)
- [x] torch.nn.init(*kophy*)(luc)
- [x] torch.optim(*ZijunDeng*)(祁杰)
- [x] torch.autograd(*KeithYin*)(祁杰)
- [x] torch.multiprocessing(*songbo.han*)
- [x] torch.legacy(*ycszen*)
- [x] torch.cuda(*ycszen*)
- [x] torch.utils.ffi(*ycszen*)
- [x] torch.utils.data(*ycszen*)
- [x] torch.utils.model_zoo(*ycszen*)

torchvision Reference

- [x] torchvision (*KeithYin*)
- [x] torchvision.datasets (*KeithYin*)(loop)
- [x] torchvision.models (*KeithYin*)
- [x] torchvision.transforms (*KeithYin*)(loop)
- [x] torchvision.utils (*KeithYin*)

PyTorch中文文档

PyTorch是使用GPU和CPU优化的深度学习张量库。

说明

- 自动求导机制
- CUDA语义
- 扩展PyTorch
- 多进程最佳实践
- 序列化语义

Package参考

- torch
- torch.Tensor
- torch.Storage
- torch.nn
- torch.nn.functional
- torch.nn.init
- torch.optim
- torch.autograd
- torch.multiprocessing
- torch.legacy
- torch.cuda
- torch.utils.ffi
- torch.utils.data
- torch.utils.model_zoo

torchvision参考

- torchvision
- torchvision.datasets
- torchvision.models
- torchvision.transforms
- torchvision.utils

致谢

自动求导机制

本说明将概述Autograd如何工作并记录操作。了解这些并不是绝对必要的，但我们建议您熟悉它，因为它将帮助您编写更高效，更简洁的程序，并可帮助您进行调试。

从后向中排除子图

每个变量都有两个标志： `requires_grad` 和 `volatile` 。它们都允许从梯度计算中精细地排除子图，并可以提高效率。

`requires_grad`

如果有一个单一的输入操作需要梯度，它的输出也需要梯度。相反，只有所有输入都不需要梯度，输出才不需要。如果其中所有的变量都不需要梯度进行，后向计算不会在子图中执行。

```
>>> x = Variable(torch.randn(5, 5))
>>> y = Variable(torch.randn(5, 5))
>>> z = Variable(torch.randn(5, 5), requires_grad=True)
>>> a = x + y
>>> a.requires_grad
False
>>> b = a + z
>>> b.requires_grad
True
```

这个标志特别有用，当您想要冻结部分模型时，或者您事先知道不会使用某些参数的梯度。例如，如果要对预先训练的CNN进行优化，只要切换冻结模型中的 `requires_grad` 标志就足够了，直到计算到最后一层才会保存中间缓冲区，其中的仿射变换将使用需要梯度的权重并且网络的输出也将需要它们。

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

`volatile`

纯粹的inference模式下推荐使用 `volatile` ，当你确定你甚至不会调用 `.backward()` 时。它比任何其他自动求导的设置更有效——它将使用绝对最小的内存来评估模型。 `volatile` 也决定了 `require_grad is False` 。

`volatile` 不同于 `require_grad` 的传递。如果一个操作甚至只有有一个 `volatile` 的输入，它的输出也将是 `volatile` 。 `Volatility` 比“不需要梯度”更容易传递——只需要一个 `volatile` 的输入即可得到一个 `volatile` 的输出，相对的，需要所有的输入“不需要梯度”才能得到不需要梯度的输出。使用volatile标志，您不需要更改模型参数的任何设置来用于inference。创建一个 `volatile` 的输入就够了，这将保证不会保存中间状态。

```
>>> regular_input = Variable(torch.randn(5, 5))
>>> volatile_input = Variable(torch.randn(5, 5), volatile=True)
>>> model = torchvision.models.resnet18(pretrained=True)
>>> model(regular_input).requires_grad
True
>>> model(volatile_input).requires_grad
False
>>> model(volatile_input).volatile
True
>>> model(volatile_input).creator is None
True
```

自动求导如何编码历史信息

每个变量都有一个 `.creator` 属性，它指向把它作为输出的函数。这是一个由 `Function` 对象作为节点组成的有向无环图（DAG）的入口点，它们之间的引用就是图的边。每次执行一个操作时，一个表示它的新 `Function` 就被实例化，它的 `forward()` 方法被调用，并且它输出的 `Variable` 的创建者被设置为这个 `Function` 。然后，通过跟踪从任何变量到叶节点的路径，可以重建创建数据的操作序列，并自动计算梯度。

需要注意的一点是，整个图在每次迭代时都是从头开始重新创建的，这就允许使用任意的Python控制流语句，这样可以在每次迭代时改变图的整体形状和大小。在启动训练之前不必对所有可能的路径进行编码—— what you run is what you differentiate.

Variable上的In-place操作

在自动求导中支持in-place操作是一件很困难的事情，我们在大多数情况下都不鼓励使用它们。Autograd的缓冲区释放和重用非常高效，并且很少场合下in-place操作能实际上明显降低内存的使用量。除非您在内存压力很大的情况下，否则您可能永远不需要使用它们。

限制in-place操作适用性主要有两个原因：

１．覆盖梯度计算所需的值。这就是为什么变量不支持 `log_` 。它的梯度公式需要原始输入，而虽然通过计算反向操作可以重新创建它，但在数值上是不稳定的，并且需要额外的工作，这往往会与使用这些功能的目的相悖。

２．每个in-place操作实际上需要实现重写计算图。不合适的版本只需分配新对象并保留对旧图的引用，而in-place操作则需要将所有输入的 `creator` 更改为表示此操作的 `Function` 。这就比较棘手，特别是如果有许多变量引用相同的存储（例如通过索引或转置创建的），并且

如果被修改输入的存储被任何其他`Variable`引用，则in-place函数实际上会抛出错误。

In-place正确性检查

每个变量保留有version counter，它每次都会递增，当在任何操作中被使用时。
当`Function`保存任何用于后向的tensor时，还会保存其包含变量的version counter。一旦访问`self.saved_tensors`，它将被检查，如果它大于保存的值，则会引起错误。

CUDA语义

`torch.cuda` 会记录当前选择的GPU，并且分配的所有CUDA张量将在上面创建。可以使用 `torch.cuda.device` 上下文管理器更改所选设备。

但是，一旦张量被分配，您可以直接对其进行操作，而不考虑所选择的设备，结果将始终放在与张量相同的设备上。

默认情况下，不支持跨GPU操作，唯一的例外是 `copy_()` 。除非启用对等存储器访问，否则对分布不同设备上的张量任何启动操作的尝试都将会引发错误。

下面你可以找到一个展示如下的小例子：

```
x = torch.cuda.FloatTensor(1)
# x.get_device() == 0
y = torch.FloatTensor(1).cuda()
# y.get_device() == 0

with torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.cuda.FloatTensor(1)

    # transfers a tensor from CPU to GPU 1
    b = torch.FloatTensor(1).cuda()
    # a.get_device() == b.get_device() == 1

    c = a + b
    # c.get_device() == 1

    z = x + y
    # z.get_device() == 0

    # even within a context, you can give a GPU id to the .cuda call
    d = torch.randn(2).cuda(2)
    # d.get_device() == 2
```

最佳实践

使用固定的内存缓冲区

当副本来自固定（页锁）内存时，主机到GPU的复制速度要快很多。CPU张量和存储开放了一个 `pin_memory()` 方法，它返回该对象的副本，而它的数据放在固定区域中。

另外，一旦固定了张量或存储，就可以使用异步的GPU副本。只需传递一个额外的 `async=True` 参数到 `cuda()` 的调用。这可以用于将数据传输与计算重叠。

通过将 `pin_memory=True` 传递给其构造函数，可以使 `DataLoader` 将batch返回到固定内存中。

使用 nn.DataParallel 替代 multiprocessing

大多数涉及批量输入和多个GPU的情况应默认使用 `DataParallel` 来使用多个GPU。尽管有GIL的存在，单个python进程也可能使多个GPU饱和。

从0.1.9版本开始，大量的GPU(8+)可能未被充分利用。然而，这是一个已知的问题，也正在积极开发。和往常一样，测试你的用例吧。

调用 `multiprocessing` 来利用CUDA模型存在重要的注意事项；使用具有多处理功能的CUDA模型有重要的注意事项; 除非就是需要谨慎地满足数据处理需求，否则您的程序很可能会出现错误或未定义的行为。

扩展PyTorch

本篇文章中包含如何扩展 `torch.nn` , `torch.autograd` 和 使用我们的 `C 库` 编写自定义的 `C` 扩展。

扩展 torch.autograd

如果你想要添加一个新的 `Operation` 到 `autograd` 的话，你的 `Operation` 需要继承 `class Function` 。 `autograd` 使用 `Function` 计算结果和梯度，同时编码 `operation` 的历史。每个新的 `operation(function)` 都需要实现三个方法：

- `__init__ (optional)` - 如果你的 `operation` 包含非 `Variable` 参数，那么就将其作为 `__init__` 的参数传入到 `operation` 中。例如： `AddConstant Function` 加一个常数， `Transpose Function` 需要指定哪两个维度需要交换。如果你的 `operation` 不需要额外的参数，你可以忽略 `__init__` 。

- `forward()` - 在里面写执行此 `operation` 的代码。可以有任意数量的参数。如果你对某些参数指定了默认值，则这些参数是可传可不传的。记住： `forward()` 的参数只能是 `Variable` 。函数的返回值既可以是 `Variable` 也可以是 `Variables` 的 `tuple` 。同时，请参考 `Function` [function]的 `doc` ，查阅有哪些 方法是只能在 `forward` 中调用的。

- `backward()` - 梯度计算公式。 参数的个数和 `forward` 返回值的个数一样，每个参数代表传回到此 `operation` 的梯度. `backward()` 的返回值的个数应该和此 `operation` 输入的个数一样，每个返回值对应了输入值的梯度。如果 `operation` 的输入不需要梯度，或者不可导，你可以返回 `None` 。 如果 `forward()` 存在可选参数，你可以返回比输入更多的梯度，只是返回的是 `None` 。

下面是 `Linear` 的实现代码：

```
# Inherit from Function
class Linear(Function):

    # bias is an optional argument
    def forward(self, input, weight, bias=None):
        self.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    def backward(self, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = self.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if self.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if self.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and self.needs_input_grad[2]:
            grad_bias = grad_output.sum(0).squeeze(0)

        return grad_input, grad_weight, grad_bias
```

现在，为了可以更简单的使用自定义的 `operation` ，我们建议将其用一个简单的 `helper function` 包装起来。 functions:

```
def linear(input, weight, bias=None):
    # First braces create a Function object. Any arguments given here
    # will be passed to __init__. Second braces will invoke the __call__
    # operator, that will then use forward() to compute the result and
    # return it.
    return Linear()(input, weight, bias)
```

你可能想知道你刚刚实现的 `backward` 方法是否正确的计算了梯度。你可以使用 小的有限的差分进行数值估计。

```
from torch.autograd import gradcheck

# gradchek takes a tuple of tensor as input, check if your gradient
# evaluated with these tensors are close enough to numerical
# approximations and returns True if they all verify this condition.
input = (Variable(torch.randn(20,20).double(), requires_grad=True),)
test = gradcheck.gradcheck(Linear(), input, eps=1e-6, atol=1e-4)
print(test)
```

扩展 torch.nn

`nn` 包含两种接口 - `modules` 和他们的 `functional` 版本。通过这两个接口，你都可以扩展 `nn` 。但是我们建议，在扩展 `layer` 的时候，使用 `modules` ，因为 `modules` 保存着参数和 `buffer` 。如果不需要参数的话，那么建议使用 `functional` (激活函数，pooling，这些都不需要参数)。

增加一个 `operation` 的 `functional` 版本已经在上面一节介绍完毕。

增加一个模块(`module`)。由于 `nn` 重度使用 `autograd` 。所以，添加一个新 `module` 需要实现一个 用来执行 计算 和 计算梯度 的 `Function` 。从现在开始，假定我们想要实现一个 `Linear module` ，记得之前我们已经实现了一个 `Linear Funciton` 。只需要很少的代码就可以完成这个工作。 现在，我们需要实现两个方法：

- `__init__ (optional)` - 输入参数，例如 `kernel sizes` , `numbers of features` , 等等。同时初始化 `parameters` 和 `buffers` 。
- `forward()` - 实例化一个执行 `operation` 的 `Function` ，使用它执行 `operation` 。和 `functional wrapper(上面实现的那个简单的wrapper)` 十分类似。

`Linear module` 实现代码:

```
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Variable, that will get
        # automatically registered as Module's parameter once it's assigned
        # as an attribute. Parameters and buffers need to be registered, or
        # they won't appear in .parameters() (doesn't apply to buffers), and
        # won't be converted when e.g. .cuda() is called. You can use
        # .register_buffer() to register buffers.
        # nn.Parameters can never be volatile and, different than Variables,
        # they require gradients by default.
        self.weight = nn.Parameter(torch.Tensor(input_features, output_features))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(output_features))
        else:
            # You should always register all possible parameters, but the
            # optional ones can be None if you want.
            self.register_parameter('bias', None)

        # Not a very smart way to initialize weights
        self.weight.data.uniform_(-0.1, 0.1)
        if bias is not None:
            self.bias.data.uniform_(-0.1, 0.1)

    def forward(self, input):
        # See the autograd section for explanation of what happens here.
        return Linear()(input, self.weight, self.bias)
        #注意这个Linear是之前实现过的Linear
```

编写自定义 `C` 扩展

Coming soon. For now you can find an example at GitHub.

多进程最佳实践

`torch.multiprocessing` 是Python `multiprocessing` 的替代品。它支持完全相同的操作，但扩展了它以便通过 `multiprocessing.Queue` 发送的所有张量将其数据移动到共享内存中，并且只会向其他进程发送一个句柄。

> **Note**
>
> 当 `Variable` 发送到另一个进程时， `Variable.data` 和 `Variable.grad.data` 都将被共享。

这允许实现各种训练方法，如Hogwild，A3C或需要异步操作的任何其他方法。

共享CUDA张量

仅在Python 3中使用 `spawn` 或 `forkserver` 启动方法才支持在进程之间共享CUDA张量。Python 2中的 `multiprocessing` 只能使用 `fork` 创建子进程，并且不被CUDA运行时所支持。

> **Warning**
>
> CUDA API要求导出到其他进程的分配，只要它们被使用就要一直保持有效。您应该小心，确保您共享的CUDA张量只要有必要就不要超出范围。这不是共享模型参数的问题，但传递其他类型的数据应该小心。注意，此限制不适用于共享CPU内存。

参考：使用 nn.DataParallel 替代 multiprocessing

最佳实践和提示

避免和抵制死锁

当一个新进程被产生时，有很多事情可能会出错，最常见的死锁原因是后台线程。如果有任何线程持有锁或导入模块，并且 `fork` 被调用，则子进程很可能处于损坏的状态，并以不同的方式死锁或失败。注意，即使您没有，Python内置的库也可能会这样做 —— 不需要看得比 `multiprocessing` 更远。 `multiprocessing.Queue` 实际上是一个非常复杂的类，它产生用于序列化，发送和接收对象的多个线程，它们也可能引起上述问题。如果您发现自己处于这种情况，请尝试使用 `multiprocessing.queues.SimpleQueue` ，这不会使用任何其他线程。

我们正在竭尽全力把它设计得更简单，并确保这些死锁不会发生，但有些事情无法控制。如果有任何问题您无法一时无法解决，请尝试在论坛上提出，我们将看看是否可以解决问题。

重用经过队列的缓冲区

记住每次将 `Tensor` 放入 `multiprocessing.Queue` 时，必须将其移动到共享内存中。如果它已经被共享，它是一个无效的操作，否则会产生一个额外的内存副本，这会减缓整个进程。即使你有一个进程池来发送数据到一个进程，使它返回缓冲区 —— 这几乎是免费的，并且允许你在发送下一个batch时避免产生副本。

异步多进程训练（例如Hogwild）

使用 `torch.multiprocessing` ，可以异步地训练模型，参数可以一直共享，也可以定期同步。在第一种情况下，我们建议发送整个模型对象，而在后者中，我们建议只发送 `state_dict()` 。

我们建议使用 `multiprocessing.Queue` 来在进程之间传递各种PyTorch对象。例如， 当使用fork启动方法时，可能会继承共享内存中的张量和存储器，但这是非常容易出错的，应谨慎使用，而且只能由高级用户使用。队列虽然有时是一个较不优雅的解决方案，但基本上能在所有情况下正常工作。

> **Warning** 你应该注意有关全局语句，它们没有被 `if __name__ == '__main__'` 保护。如果使用与 `fork` 不同的启动方法，则它们将在所有子进程中执行。

Hogwild

在examples repository中可以找到具体的Hogwild实现，可以展示代码的整体结构。下面也有一个小例子：

```
import torch.multiprocessing as mp
from model import MyModel

def train(model):
    # Construct data_loader, optimizer, etc.
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step()  # This will update the shared parameters

if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # NOTE: this is required for the ``fork`` method to work
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
      p.join()
```

序列化语义

最佳实践

保存模型的推荐方法

这主要有两种方法序列化和恢复模型。

第一种（推荐）只保存和加载模型参数：

```
torch.save(the_model.state_dict(), PATH)
```

然后：

```
the_model = TheModelClass(*args, **kwargs)
the_model.load_state_dict(torch.load(PATH))
```

第二种保存和加载整个模型：

```
torch.save(the_model, PATH)
```

然后：

```
the_model = torch.load(PATH)
```

然而，在这种情况下，序列化的数据被绑定到特定的类和固定的目录结构，所以当在其他项目中使用时，或者在一些严重的重构器之后它可能会以各种方式break。

torch

包 `torch` 包含了多维张量的数据结构以及基于其上的多种数学操作。另外，它也提供了多种工具，其中一些可以更有效地对张量和任意类型进行序列化。

它有CUDA 的对应实现，可以在NVIDIA GPU上进行张量运算(计算能力>=2.0)。

张量 Tensors

torch.is_tensor[source]

```
torch.is_tensor(obj)
```

如果*obj* 是一个pytorch张量，则返回True

- 参数： obj (Object) – 判断对象

---

torch.is_storage [source]

```
torch.is_storage(obj)
```

如何*obj* 是一个pytorch storage对象，则返回True

- 参数： input (Object) – 判断对象

---

torch.set_default_tensor_type[source]

```
torch.set_default_tensor_type(t)
```

---

torch.numel

```
torch.numel(input)->int
```

返回 `input` 张量中的元素个数

- 参数: input (*Tensor*) – 输入张量

例子:

```
>>> a = torch.randn(1,2,3,4,5)
>>> torch.numel(a)
120
>>> a = torch.zeros(4,4)
>>> torch.numel(a)
16
```

torch.set_printoptions[source]

```
torch.set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=None,
 profile=None)
```

设置打印选项。 完全参考自 Numpy。

参数:

- precision – 浮点数输出的精度位数 (默认为8)
- threshold – 阈值，触发汇总显示而不是完全显示(repr)的数组元素的总数 （默认为1000）
- edgeitems – 汇总显示中，每维（轴）两端显示的项数（默认值为3）
- linewidth – 用于插入行间隔的每行字符数（默认为80）。Thresholded matricies will ignore this parameter.
- profile – pretty打印的完全默认值。 可以覆盖上述所有选项 (默认为short, full)

创建操作 Creation Ops

torch.eye

```
torch.eye(n, m=None, out=None)
```

返回一个2维张量，对角线位置全1，其它位置全0

参数:

- n (int) – 行数
- m (int, *optional*) – 列数.如果为None,则默认为*n*
- out (*Tensor*, *optinal*) - Output tensor

返回值: 对角线位置全1，其它位置全0的2维张量

返回值类型: Tensor

例子:

```
>>> torch.eye(3)
 1  0  0
 0  1  0
 0  0  1
[torch.FloatTensor of size 3x3]
```

---

from_numpy

```
torch.from_numpy(ndarray) → Tensor
```

Numpy桥，将 `numpy.ndarray` 转换为pytorch的 `Tensor` 。 返回的张量tensor和numpy的ndarray共享同一内存空间。修改一个会导致另外一个也被修改。返回的张量不能改变大小。

例子:

```
>>> a = numpy.array([1, 2, 3])
>>> t = torch.from_numpy(a)
>>> t
torch.LongTensor([1, 2, 3])
>>> t[0] = -1
>>> a
array([-1,  2,  3])
```

torch.linspace

```
torch.linspace(start, end, steps=100, out=None) → Tensor
```

返回一个1维张量，包含在区间 `start` 和 `end` 上均匀间隔的 `steps` 个点。 输出1维张量的长度为 `steps` 。

参数:

- start (float) – 序列的起始点
- end (float) – 序列的最终值
- steps (int) – 在 `start` 和 `end` 间生成的样本数
- out (Tensor, optional) – 结果张量

例子:

```
>>> torch.linspace(3, 10, steps=5)

  3.0000
  4.7500
  6.5000
  8.2500
 10.0000
[torch.FloatTensor of size 5]

>>> torch.linspace(-10, 10, steps=5)

-10
 -5
  0
  5
 10
[torch.FloatTensor of size 5]

>>> torch.linspace(start=-10, end=10, steps=5)

-10
 -5
  0
  5
 10
[torch.FloatTensor of size 5]
```

---

torch.logspace

```
torch.logspace(start, end, steps=100, out=None) → Tensor
```

返回一个1维张量，包含在区间 10^{start} 和 10^{end} 上以对数刻度均匀间隔的 `steps` 个点。 输出1维张量的长度为 `steps` 。

参数:

- start (float) – 序列的起始点
- end (float) – 序列的最终值
- steps (int) – 在 `start` 和 `end` 间生成的样本数
- out (Tensor, optional) – 结果张量

例子:

```
>>> torch.logspace(start=-10, end=10, steps=5)

 1.0000e-10
 1.0000e-05
 1.0000e+00
 1.0000e+05
 1.0000e+10
[torch.FloatTensor of size 5]

>>> torch.logspace(start=0.1, end=1.0, steps=5)

  1.2589
  2.1135
  3.5481
  5.9566
 10.0000
[torch.FloatTensor of size 5]
```

torch.ones

```
torch.ones(*sizes, out=None) → Tensor
```

返回一个全为1 的张量，形状由可变参数 `sizes` 定义。

参数:

- sizes (int...) – 整数序列，定义了输出形状
- out (Tensor, optional) – 结果张量 例子: ```python

  > > > torch.ones(2, 3)

  1 1 1 1 1 1 [torch.FloatTensor of size 2x3]

> > > torch.ones(5)

1 1 1 1 1 [torch.FloatTensor of size 5]

```
***

**  torch.rand**
```python
torch.rand(*sizes, out=None) → Tensor
```

返回一个张量，包含了从区间[0,1)的均匀分布中抽取的一组随机数，形状由可变参数 `sizes` 定义。

参数:

- sizes (int...) – 整数序列，定义了输出形状
- out (*Tensor*, *optinal*) - 结果张量 例子： ```python

  > > > torch.rand(4)
```

0.9193 0.3347 0.3232 0.7715 [torch.FloatTensor of size 4]

> > > torch.rand(2, 3)

0.5010 0.5140 0.0719 0.1435 0.5636 0.0538 [torch.FloatTensor of size 2x3]

```
***

** torch.randn**
```python
torch.randn(*sizes, out=None) → Tensor
```

返回一个张量，包含了从标准正态分布(均值为0，方差为 1，即高斯白噪声)中抽取一组随机数，形状由可变参数 `sizes` 定义。 参数:

- sizes (int...) – 整数序列，定义了输出形状
- out (*Tensor*, *optinal*) - 结果张量

例子：:

```
>>> torch.randn(4)

-0.1145
 0.0094
-1.1717
 0.9846
[torch.FloatTensor of size 4]

>>> torch.randn(2, 3)

 1.4339  0.3351 -1.0999
 1.5458 -0.9643 -0.3558
[torch.FloatTensor of size 2x3]
```

---

torch.randperm

```
torch.randperm(n, out=None) → LongTensor
```

给定参数 `n` ，返回一个从 `0` 到 `n -1` 的随机整数排列。

参数:

- n (int) – 上边界(不包含)

例子：
```

```
>>> torch.randperm(4)

 2
 1
 3
 0
[torch.LongTensor of size 4]
```

---

torch.arange

```
torch.arange(start, end, step=1, out=None) → Tensor
```

返回一个1维张量，长度为 \(floor((end−start)/step) \)。包含从 `start` 到 `end` ，以 `step` 为步长的一组序列值(默认步长为1)。

参数:

- start (float) – 序列的起始点
- end (float) – 序列的终止点
- step (float) – 相邻点的间隔大小
- out (Tensor, optional) – 结果张量

例子：

```
>>> torch.arange(1, 4)

 1
 2
 3
[torch.FloatTensor of size 3]

>>> torch.arange(1, 2.5, 0.5)

 1.0000
 1.5000
 2.0000
[torch.FloatTensor of size 3]
```

---

torch.range

```
torch.range(start, end, step=1, out=None) → Tensor
```

返回一个1维张量，有 \(floor((end−start)/step)+1 \) 个元素。包含在半开区间 `[start, end）` 从 `start` 开始，以 `step` 为步长的一组值。 `step` 是两个值之间的间隔，即 \(x_{i+1}=x_i+step \)

警告：建议使用函数 `torch.arange()`

参数:

- start (float) – 序列的起始点
- end (float) – 序列的最终值
- step (int) – 相邻点的间隔大小
- out (Tensor, optional) – 结果张量

例子：

```
>>> torch.range(1, 4)

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> torch.range(1, 4, 0.5)

 1.0000
 1.5000
 2.0000
 2.5000
 3.0000
 3.5000
 4.0000
[torch.FloatTensor of size 7]
```

---

torch.zeros

```
torch.zeros(*sizes, out=None) → Tensor
```

返回一个全为标量 0 的张量，形状由可变参数 `sizes` 定义。

参数:

- sizes (int...) – 整数序列，定义了输出形状
- out (Tensor, *optional*) – 结果张量

例子：

```
>>> torch.zeros(2, 3)

 0  0  0
 0  0  0
[torch.FloatTensor of size 2x3]

>>> torch.zeros(5)

 0
 0
 0
 0
 0
[torch.FloatTensor of size 5]
```

索引,切片,连接,换位Indexing, Slicing, Joining, Mutating Ops

torch.cat

```
torch.cat(inputs, dimension=0) → Tensor
```

在给定维度上对输入的张量序列 `seq` 进行连接操作。

`torch.cat()` 可以看做 `torch.split()` 和 `torch.chunk()` 的反操作。 `cat()` 函数可以通过下面例子更好的理解。

参数:

- inputs (*sequence of Tensors*) – 可以是任意相同Tensor 类型的python 序列
- dimension (*int*, *optional*) – 沿着此维连接张量序列。

  例子： ```python

  > > > x = torch.randn(2, 3) x

  0.5983 -0.0341 2.4918 1.5981 -0.5265 -0.8735 [torch.FloatTensor of size 2x3]

> > > torch.cat((x, x, x), 0)

0.5983 -0.0341 2.4918 1.5981 -0.5265 -0.8735 0.5983 -0.0341 2.4918 1.5981 -0.5265 -0.8735 0.5983 -0.0341 2.4918 1.5981 -0.5265 -0.8735 [torch.FloatTensor of size 6x3]

> > > torch.cat((x, x, x), 1)

0.5983 -0.0341 2.4918 0.5983 -0.0341 2.4918 0.5983 -0.0341 2.4918 1.5981 -0.5265 -0.8735 1.5981 -0.5265 -0.8735 1.5981 -0.5265 -0.8735 [torch.FloatTensor of size 2x9]

```
### torch.chunk
```python
torch.chunk(tensor, chunks, dim=0)
```

在给定维度(轴)上将输入张量进行分块儿。

参数:

- tensor (Tensor) – 待分块的输入张量
- chunks (int) – 分块的个数
- dim (int) – 沿着此维度进行分块

torch.gather

```
torch.gather(input, dim, index, out=None) → Tensor
```

沿给定轴 `dim` ，将输入索引张量 `index` 指定位置的值进行聚合。

对一个3维张量，输出可以定义为：

```
out[i][j][k] = tensor[index[i][j][k]][j][k]  # dim=0
out[i][j][k] = tensor[i][index[i][j][k]][k]  # dim=1
out[i][j][k] = tensor[i][j][index[i][j][k]]  # dim=3
```

例子：

```
>>> t = torch.Tensor([[1,2],[3,4]])
>>> torch.gather(t, 1, torch.LongTensor([[0,0],[1,0]]))
 1  1
 4  3
[torch.FloatTensor of size 2x2]
```

参数:

- input (Tensor) – 源张量
- dim (int) – 索引的轴
- index (LongTensor) – 聚合元素的下标
- out (Tensor, optional) – 目标张量

torch.index_select

```
torch.index_select(input, dim, index, out=None) → Tensor
```
```

沿着指定维度对输入进行切片，取 `index` 中指定的相应项(`index` 为一个LongTensor)，然后返回到一个新的张量， 返回的张量与原始张量*Tensor*有相同的维度(在指定轴上)。

注意： 返回的张量不与原始张量共享内存空间。

参数:

- input (Tensor) – 输入张量
- dim (int) – 索引的轴
- index (LongTensor) – 包含索引下标的一维张量
- out (Tensor, optional) – 目标张量

例子：

```
>>> x = torch.randn(3, 4)
>>> x

 1.2045  2.4084  0.4001  1.1372
 0.5596  1.5677  0.6219 -0.7954
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 3x4]

>>> indices = torch.LongTensor([0, 2])
>>> torch.index_select(x, 0, indices)

 1.2045  2.4084  0.4001  1.1372
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 2x4]

>>> torch.index_select(x, 1, indices)

 1.2045  0.4001
 0.5596  0.6219
 1.3635 -0.5414
[torch.FloatTensor of size 3x2]
```

torch.masked_select

```
torch.masked_select(input, mask, out=None) → Tensor
```

根据掩码张量 `mask` 中的二元值，取输入张量中的指定项(`mask` 为一个 *ByteTensor*)，将取值返回到一个新的1D张量，

张量 `mask` 须跟 `input` 张量有相同数量的元素数目，但形状或维度不需要相同。 注意： 返回的张量不与原始张量共享内存空间。

参数:

- input (Tensor) – 输入张量
- mask (ByteTensor) – 掩码张量，包含了二元索引值
- out (Tensor, optional) – 目标张量

例子：

```
>>> x = torch.randn(3, 4)
>>> x

 1.2045  2.4084  0.4001  1.1372
 0.5596  1.5677  0.6219 -0.7954
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 3x4]

>>> indices = torch.LongTensor([0, 2])
>>> torch.index_select(x, 0, indices)

 1.2045  2.4084  0.4001  1.1372
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 2x4]

>>> torch.index_select(x, 1, indices)

 1.2045  0.4001
 0.5596  0.6219
 1.3635 -0.5414
[torch.FloatTensor of size 3x2]
```

torch.nonzero

```
torch.nonzero(input, out=None) → LongTensor
```

返回一个包含输入 `input` 中非零元素索引的张量。输出张量中的每行包含输入中非零元素的索引。

如果输入 `input` 有 `n` 维，则输出的索引张量 `output` 的形状为 z x n, 这里 z 是输入张量 `input` 中所有非零元素的个数。

参数:

- input (Tensor) – 源张量
- out (LongTensor, optional) – 包含索引值的结果张量

例子：

```
>>> torch.nonzero(torch.Tensor([1, 1, 1, 0, 1]))

 0
 1
 2
 4
[torch.LongTensor of size 4x1]

>>> torch.nonzero(torch.Tensor([[0.6, 0.0, 0.0, 0.0],
...                             [0.0, 0.4, 0.0, 0.0],
...                             [0.0, 0.0, 1.2, 0.0],
...                             [0.0, 0.0, 0.0,-0.4]]))

 0  0
 1  1
 2  2
 3  3
[torch.LongTensor of size 4x2]
```

torch.split

```
torch.split(tensor, split_size, dim=0)
```

将输入张量分割成相等形状的chunks（如果可分）。 如果沿指定维的张量形状大小不能被 `split_size` 整分， 则最后一个分块会小于其它分块。

参数:

- tensor (Tensor) – 待分割张量
- split_size (int) – 单个分块的形状大小
- dim (int) – 沿着此维进行分割

torch.squeeze

```
torch.squeeze(input, dim=None, out=None)
```

将输入张量形状中的 `1` 去除并返回。 如果输入是形如\((A \times 1\times B \times 1 \times C \times 1 \times D) \)，那么输出形状就为： \((A \times B \times C \times D) \)

当给定 `dim` 时，那么挤压操作只在给定维度上。例如，输入形状为: \((A \times 1 \times B) \), `squeeze(input, 0)` 将会保持张量不变，只有用 `squeeze(input, 1)`，形状会变成 \((A \times B)\)。

注意： 返回张量与输入张量共享内存，所以改变其中一个的内容会改变另一个。

参数:

- input (Tensor) – 输入张量
- dim (int, optional) – 如果给定，则 `input` 只会在给定维度挤压

- out (Tensor, optional) – 输出张量

例子：

```
>>> x = torch.zeros(2,1,2,1,2)
>>> x.size()
(2L, 1L, 2L, 1L, 2L)
>>> y = torch.squeeze(x)
>>> y.size()
(2L, 2L, 2L)
>>> y = torch.squeeze(x, 0)
>>> y.size()
(2L, 1L, 2L, 1L, 2L)
>>> y = torch.squeeze(x, 1)
>>> y.size()
(2L, 2L, 1L, 2L)
```

torch.stack[source]

```
torch.stack(sequence, dim=0)
```

沿着一个新维度对输入张量序列进行连接。 序列中所有的张量都应该为相同形状。

参数:

- sqequence (Sequence) – 待连接的张量序列
- dim (int) – 插入的维度。必须介于 0 与 待连接的张量序列数之间。

torch.t

```
torch.t(input, out=None) → Tensor
```

输入一个矩阵（2维张量），并转置0, 1维。 可以被视为函数`transpose(input, 0, 1)`的简写函数。

参数:

- input (Tensor) – 输入张量
- out (Tensor, optional) – 结果张量 ```python

  > > > x = torch.randn(2, 3) x

  0.4834 0.6907 1.3417 -0.1300 0.5295 0.2321 [torch.FloatTensor of size 2x3]

> > > torch.t(x)

0.4834 -0.1300 0.6907 0.5295 1.3417 0.2321 [torch.FloatTensor of size 3x2]

```
### torch.transpose
```python
torch.transpose(input, dim0, dim1, out=None) → Tensor
```

返回输入矩阵 `input` 的转置。交换维度 `dim0` 和 `dim1` 。 输出张量与输入张量共享内存，所以改变其中一个会导致另外一个也被修改。

参数:

- input (Tensor) – 输入张量
- dim0 (int) – 转置的第一维
- dim1 (int) – 转置的第二维

```
>>> x = torch.randn(2, 3)
>>> x

 0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735
[torch.FloatTensor of size 2x3]

>>> torch.transpose(x, 0, 1)

 0.5983  1.5981
-0.0341 -0.5265
 2.4918 -0.8735
[torch.FloatTensor of size 3x2]
```

torch.unbind

```
torch.unbind(tensor, dim=0)[source]
```

移除指定维后，返回一个元组，包含了沿着指定维切片后的各个切片

参数:

- tensor (Tensor) – 输入张量
- dim (int) – 删除的维度

torch.unsqueeze

```
torch.unsqueeze(input, dim, out=None)
```

返回一个新的张量，对输入的制定位置插入维度 1

注意： 返回张量与输入张量共享内存，所以改变其中一个的内容会改变另一个。

如果 `dim` 为负，则将会被转化\(dim+input.dim()+1 \)
```

参数:

- tensor (Tensor) – 输入张量
- dim (int) – 插入维度的索引
- out (Tensor, optional) – 结果张量

```
>>> x = torch.Tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
 1  2  3  4
[torch.FloatTensor of size 1x4]
>>> torch.unsqueeze(x, 1)
 1
 2
 3
 4
[torch.FloatTensor of size 4x1]
```

随机抽样 Random sampling

torch.manual_seed

```
torch.manual_seed(seed)
```

设定生成随机数的种子，并返回一个 *torch._C.Generator* 对象.

参数: seed (int or long) – 种子.

torch.initial_seed

```
torch.initial_seed()
```

返回生成随机数的原始种子值（python long）。

torch.get_rng_state

```
torch.get_rng_state()[source]
```

返回随机生成器状态(*ByteTensor*)

torch.set_rng_state

```
torch.set_rng_state(new_state)[source]
```

设定随机生成器状态 参数: new_state (torch.ByteTensor) – 期望的状态

torch.default_generator

```
torch.default_generator = <torch._C.Generator object>
```

torch.bernoulli

```
torch.bernoulli(input, out=None) → Tensor
```

从伯努利分布中抽取二元随机数(0 或者 1)。

输入张量须包含用于抽取上述二元随机值的概率。 因此，输入中的所有值都必须在［0,1］区间，即 \(0<=input_i<=1 \)

输出张量的第 `i` 个元素值， 将会以输入张量的第 `i` 个概率值等于 `1` 。

返回值将会是与输入相同大小的张量，每个值为0或者1 参数:

- input (Tensor) – 输入为伯努利分布的概率值
- out (Tensor, optional) – 输出张量(可选)

例子：

```
>>> a = torch.Tensor(3, 3).uniform_(0, 1) # generate a uniform random matrix with rang
e [0, 1]
>>> a

 0.7544  0.8140  0.9842
 0.5282  0.0595  0.6445
 0.1925  0.9553  0.9732
[torch.FloatTensor of size 3x3]

>>> torch.bernoulli(a)

 1  1  1
 0  0  1
 0  1  1
[torch.FloatTensor of size 3x3]

>>> a = torch.ones(3, 3) # probability of drawing "1" is 1
>>> torch.bernoulli(a)

 1  1  1
 1  1  1
 1  1  1
[torch.FloatTensor of size 3x3]

>>> a = torch.zeros(3, 3) # probability of drawing "1" is 0
>>> torch.bernoulli(a)

 0  0  0
 0  0  0
 0  0  0
[torch.FloatTensor of size 3x3]
```

torch.multinomial

```
torch.multinomial(input, num_samples,replacement=False, out=None) → LongTensor
```

返回一个张量，每行包含从 `input` 相应行中定义的多项分布中抽取的 `num_samples` 个样本。

[注意]:输入 `input` 每行的值不需要总和为1 (这里我们用来做权重)，但是必须非负且总和不能为0。

当抽取样本时，依次从左到右排列(第一个样本对应第一列)。

如果输入 `input` 是一个向量，输出 `out` 也是一个相同长度 `num_samples` 的向量。如果输入 `input` 是有 \(m \)行的矩阵，输出 `out` 是形如\(m \times n \)的矩阵。

如果参数 `replacement` 为 *True*, 则样本抽取可以重复。否则，一个样本在每行不能被重复抽取。

参数 `num_samples` 必须小于 `input` 长度(即， `input` 的列数，如果是 `input` 是一个矩阵)。

参数:

- input (Tensor) – 包含概率值的张量
- num_samples (int) – 抽取的样本数
- replacement (bool, optional) – 布尔值，决定是否能重复抽取
- out (Tensor, optional) – 结果张量

例子：

```
>>> weights = torch.Tensor([0, 10, 3, 0]) # create a Tensor of weights
>>> torch.multinomial(weights, 4)

 1
 2
 0
 0
[torch.LongTensor of size 4]

>>> torch.multinomial(weights, 4, replacement=True)

 1
 2
 1
 2
[torch.LongTensor of size 4]
```

torch.normal()

```
torch.normal(means, std, out=None)
```

返回一个张量，包含从给定参数 `means` , `std` 的离散正态分布中抽取随机数。 均值 `means` 是一个张量，包含每个输出元素相关的正态分布的均值。 `std` 是一个张量，包含每个输出元素相关的正态分布的标准差。 均值和标准差的形状不须匹配，但每个张量的元素个数须相同。

参数:

- means (Tensor) – 均值
- std (Tensor) – 标准差
- out (Tensor) – 可选的输出张量

```
torch.normal(means=torch.arange(1, 11), std=torch.arange(1, 0, -0.1))

 1.5104
 1.6955
 2.4895
 4.9185
 4.9895
 6.9155
 7.3683
 8.1836
 8.7164
 9.8916
[torch.FloatTensor of size 10]
```

```
torch.normal(mean=0.0, std, out=None)
```

与上面函数类似，所有抽取的样本共享均值。

参数:

- means (Tensor,optional) – 所有分布均值
- std (Tensor) – 每个元素的标准差
- out (Tensor) – 可选的输出张量

例子:

```
>>> torch.normal(mean=0.5, std=torch.arange(1, 6))

  0.5723
  0.0871
 -0.3783
 -2.5689
 10.7893
[torch.FloatTensor of size 5]
```

```
torch.normal(means, std=1.0, out=None)
```

与上面函数类似，所有抽取的样本共享标准差。

参数:

- means (Tensor) – 每个元素的均值
- std (float, optional) – 所有分布的标准差
- out (Tensor) – 可选的输出张量

例子:

```
>>> torch.normal(means=torch.arange(1, 6))

 1.1681
 2.8884
 3.7718
 2.5616
 4.2500
[torch.FloatTensor of size 5]
```

序列化 Serialization

torch.saves[source]

```
torch.save(obj, f, pickle_module=<module 'pickle' from '/home/jenkins/miniconda/lib/python3.5/pickle.py'>, pickle_protocol=2)
```

保存一个对象到一个硬盘文件上 参考: Recommended approach for saving a model 参数：

- obj – 保存对象
- f － 类文件对象 (返回文件描述符)或一个保存文件名的字符串
- pickle_module – 用于pickling元数据和对象的模块
- pickle_protocol – 指定pickle protocal 可以覆盖默认参数

torch.load[source]

```
torch.load(f, map_location=None, pickle_module=<module 'pickle' from '/home/jenkins/miniconda/lib/python3.5/pickle.py'>)
```

从磁盘文件中读取一个通过 `torch.save()` 保存的对象。 `torch.load()` 可通过参数 `map_location` 动态地进行内存重映射，使其能从不动设备中读取文件。一般调用时，需两个参数: storage 和 location tag. 返回不同地址中的storage，或着返回None (此时地址可以通过默认方法进行解析). 如果这个参数是字典的话，意味着其是从文件的地址标记到当前系统的地址标记的映射。 默认情况下， location tags中 "cpu"对应host tensors，‘cuda:device_id’ (e.g. ‘cuda:2’) 对应cuda tensors。 用户可以通过register_package进行扩展，使用自己定义的标记和反序列化方法。

参数:

- f – 类文件对象 (返回文件描述符)或一个保存文件名的字符串
- map_location – 一个函数或字典规定如何remap存储位置
- pickle_module – 用于unpickling元数据和对象的模块 (必须匹配序列化文件时的 pickle_module)

例子:

```
>>> torch.load('tensors.pt')
# Load all tensors onto the CPU
>>> torch.load('tensors.pt', map_location=lambda storage, loc: storage)
# Map tensors from GPU 1 to GPU 0
>>> torch.load('tensors.pt', map_location={'cuda:1':'cuda:0'})
```

并行化 Parallelism

torch.get_num_threads

```
torch.get_num_threads() → int
```

获得用于并行化CPU操作的OpenMP线程数

---

torch.set_num_threads

```
torch.set_num_threads(int)
```

设定用于并行化CPU操作的OpenMP线程数

数学操作Math operations

Pointwise Ops

torch.abs

```
torch.abs(input, out=None) → Tensor
```

计算输入张量的每个元素绝对值

例子：

```
>>> torch.abs(torch.FloatTensor([-1, -2, 3]))
FloatTensor([1, 2, 3])
```

torch.acos(input, out=None) → Tensor

```
torch.acos(input, out=None) → Tensor
```

返回一个新张量，包含输入张量每个元素的反余弦。 参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4)
>>> a

-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.acos(a)
 2.2608
 1.2956
 1.1075
    nan
[torch.FloatTensor of size 4]
```

torch.add()

```
torch.add(input, value, out=None)
```

对输入张量 `input` 逐元素加上标量值 `value` ，并返回结果到一个新的张量 `out` ，即 \(out = tensor + value \)。

如果输入 `input` 是FloatTensor or DoubleTensor类型，则 `value` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型， `value` 取整数、实数皆可。】

- input (Tensor) – 输入张量
- value (Number) – 添加到输入每个元素的数
- out (Tensor, optional) – 结果张量

```
>>> a = torch.randn(4)
>>> a

 0.4050
-1.2227
 1.8688
-0.4185
[torch.FloatTensor of size 4]

>>> torch.add(a, 20)

 20.4050
 18.7773
 21.8688
 19.5815
[torch.FloatTensor of size 4]
```

```
torch.add(input, value=1, other, out=None)
```

`other` 张量的每个元素乘以一个标量值 `value` ，并加到 `input` 张量上。返回结果到输出张量 `out` 。即，\(out=input+(other∗value) \)

两个张量 `input` and `other` 的尺寸不需要匹配，但元素总数必须一样。

注意 :当两个张量形状不匹配时，输入张量的形状会作为输出张量的尺寸。

如果 `other` 是FloatTensor or DoubleTensor类型，则 `value` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型， `value` 取整数、实数皆可。】

参数:

- input (Tensor) – 第一个输入张量
- value (Number) – 用于第二个张量的尺寸因子
- other (Tensor) – 第二个输入张量
- out (Tensor, optional) – 结果张量

例子：

```
>>> import torch
>>> a = torch.randn(4)
>>> a

-0.9310
 2.0330
 0.0852
-0.2941
[torch.FloatTensor of size 4]

>>> b = torch.randn(2, 2)
>>> b

 1.0663  0.2544
-0.1513  0.0749
[torch.FloatTensor of size 2x2]

>>> torch.add(a, 10, b)
 9.7322
 4.5770
-1.4279
 0.4552
[torch.FloatTensor of size 4]
```

torch.addcdiv

```
torch.addcdiv(tensor, value=1, tensor1, tensor2, out=None) → Tensor
```

用 `tensor2` 对 `tensor1` 逐元素相除，然后乘以标量值 `value` 并加到 `tensor` 。

张量的形状不需要匹配，但元素数量必须一致。

如果输入是FloatTensor or DoubleTensor类型，则 `value` 必须为实数，否则须为整数。

参数：

- tensor (Tensor) – 张量，对 tensor1 ./ tensor 进行相加
- value (Number, optional) – 标量，对 tensor1 ./ tensor2 进行相乘
- tensor1 (Tensor) – 张量，作为被除数(分子)
- tensor2 (Tensor) –张量，作为除数(分母)
- out (Tensor, optional) – 输出张量

例子：

```
>>> t = torch.randn(2, 3)
>>> t1 = torch.randn(1, 6)
>>> t2 = torch.randn(6, 1)
>>> torch.addcdiv(t, 0.1, t1, t2)

 0.0122 -0.0188 -0.2354
 0.7396 -1.5721  1.2878
[torch.FloatTensor of size 2x3]
```

torch.addcmul

```
torch.addcmul(tensor, value=1, tensor1, tensor2, out=None) → Tensor
```

用 `tensor2` 对 `tensor1` 逐元素相乘，并对结果乘以标量值 `value` 然后加到 `tensor` 。 张量的形状不需要匹配，但元素数量必须一致。 如果输入是FloatTensor or DoubleTensor类型，则 `value` 必须为实数，否则须为整数。

参数：

- tensor (Tensor) – 张量，对tensor1 ./ tensor 进行相加
- value (Number, optional) – 标量，对 tensor1 . tensor2 进行相乘
- tensor1 (Tensor) – 张量，作为乘子1
- tensor2 (Tensor) –张量，作为乘子2
- out (Tensor, optional) – 输出张量

例子：

```
>>> t = torch.randn(2, 3)
>>> t1 = torch.randn(1, 6)
>>> t2 = torch.randn(6, 1)
>>> torch.addcmul(t, 0.1, t1, t2)

 0.0122 -0.0188 -0.2354
 0.7396 -1.5721  1.2878
[torch.FloatTensor of size 2x3]
```

torch.asin

```
torch.asin(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的反正弦函数

参数：

- tensor (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.asin(a)
-0.6900
 0.2752
 0.4633
    nan
[torch.FloatTensor of size 4]
```

torch.atan

```
torch.atan(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的反正切函数

参数：

- tensor (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.atan(a)
-0.5669
 0.2653
 0.4203
 0.9196
[torch.FloatTensor of size 4]
```

torch.atan2

```
torch.atan2(input1, input2, out=None) → Tensor
```

返回一个新张量，包含两个输入张量 `input1` 和 `input2` 的反正切函数

参数：

- input1 (Tensor) – 第一个输入张量
- input2 (Tensor) – 第二个输入张量

- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.atan2(a, torch.randn(4))
-2.4167
 2.9755
 0.9363
 1.6613
[torch.FloatTensor of size 4]
```

torch.ceil

```
torch.ceil(input, out=None) → Tensor
```

天井函数，对输入 `input` 张量每个元素向上取整, 即取不小于每个元素的最小整数，并返回结果到输出。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.ceil(a)

 2
 1
-0
-0
[torch.FloatTensor of size 4]
```

torch.clamp

```
torch.clamp(input, min, max, out=None) → Tensor
```

将输入 `input` 张量每个元素的夹紧到区间 \([min, max] \)，并返回结果到一个新张量。

操作定义如下：

```
      | min, if x_i < min
y_i = | x_i, if min <= x_i <= max
      | max, if x_i > max
```

如果输入是FloatTensor or DoubleTensor类型，则参数 `min` `max` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型，`min`，`max` 取整数、实数皆可。】

参数：

- input (Tensor) – 输入张量
- min (Number) – 限制范围下限
- max (Number) – 限制范围上限
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.clamp(a, min=-0.5, max=0.5)

 0.5000
 0.3912
-0.5000
-0.5000
[torch.FloatTensor of size 4]
```

```
torch.clamp(input, *, min, out=None) → Tensor
```

将输入 `input` 张量每个元素的限制到不小于 `min` ，并返回结果到一个新张量。

如果输入是FloatTensor or DoubleTensor类型，则参数 `min` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型，`min` 取整数、实数皆可。】

参数：

- input (Tensor) – 输入张量
- value (Number) – 限制范围下限
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.clamp(a, min=0.5)

 1.3869
 0.5000
 0.5000
 0.5000
[torch.FloatTensor of size 4]
```

```
torch.clamp(input, *, max, out=None) → Tensor
```

将输入 `input` 张量每个元素的限制到不大于 `max` ，并返回结果到一个新张量。

如果输入是FloatTensor or DoubleTensor类型，则参数 `max` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型， `max` 取整数、实数皆可。】

参数：

- input (Tensor) – 输入张量
- value (Number) – 限制范围上限
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.clamp(a, max=0.5)

 0.5000
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]
```

torch.cos

```
torch.cos(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的余弦。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.cos(a)
 0.8041
 0.9633
 0.9018
 0.2557
[torch.FloatTensor of size 4]
```

torch.cosh

```
torch.cosh(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的双曲余弦。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.cosh(a)
 1.2095
 1.0372
 1.1015
 1.9917
[torch.FloatTensor of size 4]
```

torch.div()

```
torch.div(input, value, out=None)
```

将 `input` 逐元素除以标量值 `value` ，并返回结果到输出张量 `out` 。 即 \(out=tensor/value \)

如果输入是FloatTensor or DoubleTensor类型，则参数 `value` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型， `value` 取整数、实数皆可。】

参数：

- input (Tensor) – 输入张量
- value (Number) – 除数
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(5)
>>> a

-0.6147
-1.1237
-0.1604
-0.6853
 0.1063
[torch.FloatTensor of size 5]

>>> torch.div(a, 0.5)

-1.2294
-2.2474
-0.3208
-1.3706
 0.2126
[torch.FloatTensor of size 5]
```

```
torch.div(input, other, out=None)
```

两张量 `input` 和 `other` 逐元素相除，并将结果返回到输出。即， \(out_i= input_i / other_i \)

两张量形状不须匹配，但元素数须一致。

注意：当形状不匹配时， `input` 的形状作为输出张量的形状。

参数：

- input (Tensor) – 张量(分子)
- other (Tensor) – 张量(分母)
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4,4)
>>> a

-0.1810  0.4017  0.2863 -0.1013
 0.6183  2.0696  0.9012 -1.5933
 0.5679  0.4743 -0.0117 -0.1266
-0.1213  0.9629  0.2682  1.5968
[torch.FloatTensor of size 4x4]

>>> b = torch.randn(8, 2)
>>> b

 0.8774  0.7650
 0.8866  1.4805
-0.6490  1.1172
 1.4259 -0.8146
 1.4633 -0.1228
 0.4643 -0.6029
 0.3492  1.5270
 1.6103 -0.6291
[torch.FloatTensor of size 8x2]

>>> torch.div(a, b)

-0.2062  0.5251  0.3229 -0.0684
-0.9528  1.8525  0.6320  1.9559
 0.3881 -3.8625 -0.0253  0.2099
-0.3473  0.6306  0.1666 -2.5381
[torch.FloatTensor of size 4x4]
```

torch.exp

```
torch.exp(tensor, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的指数。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

  ```
  >>> torch.exp(torch.Tensor([0, math.log(2)]))
  torch.FloatTensor([1, 2])
  ```

torch.floor

```
torch.floor(input, out=None) → Tensor
```

床函数: 返回一个新张量，包含输入 `input` 张量每个元素的floor，即不小于元素的最大整数。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.floor(a)

 1
 0
-1
-1
[torch.FloatTensor of size 4]
```

torch.fmod

```
torch.fmod(input, divisor, out=None) → Tensor
```

计算除法余数。 除数与被除数可能同时含有整数和浮点数。此时，余数的正负与被除数相同。

参数：

- input (Tensor) – 被除数
- divisor (Tensor or float) – 除数，一个数或与被除数相同类型的张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> torch.fmod(torch.Tensor([-3, -2, -1, 1, 2, 3]), 2)
torch.FloatTensor([-1, -0, -1, 1, 0, 1])
>>> torch.fmod(torch.Tensor([1, 2, 3, 4, 5]), 1.5)
torch.FloatTensor([1.0, 0.5, 0.0, 1.0, 0.5])
```

参考: `torch.remainder()`, 计算逐元素余数， 相当于python 中的 % 操作符。

torch.frac

```
torch.frac(tensor, out=None) → Tensor
```

返回每个元素的分数部分。

例子：

```
>>> torch.frac(torch.Tensor([1, 2.5, -3.2])
torch.FloatTensor([0, 0.5, -0.2])
```

torch.lerp

```
torch.lerp(start, end, weight, out=None)
```

对两个张量以 `start` ， `end` 做线性插值， 将结果返回到输出张量。

即，\(out_i=start_i+weight∗(end_i−start_i) \)

参数：

- start (Tensor) – 起始点张量
- end (Tensor) – 终止点张量
- weight (float) – 插值公式的weight
- out (Tensor, optional) – 结果张量

例子：

```
>>> start = torch.arange(1, 5)
>>> end = torch.Tensor(4).fill_(10)
>>> start

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> end

 10
 10
 10
 10
[torch.FloatTensor of size 4]

>>> torch.lerp(start, end, 0.5)

 5.5000
 6.0000
 6.5000
 7.0000
[torch.FloatTensor of size 4]
```

torch.log

```
torch.log(input, out=None) → Tensor
```

计算 `input` 的自然对数

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(5)
>>> a

-0.4183
 0.3722
-0.3091
 0.4149
 0.5857
[torch.FloatTensor of size 5]

>>> torch.log(a)

    nan
-0.9883
    nan
-0.8797
-0.5349
[torch.FloatTensor of size 5]
```

torch.log1p

```
torch.log1p(input, out=None) → Tensor
```

计算 \(input +1 \)的自然对数 \(y_i=log(x_i+1) \)

注意：对值比较小的输入，此函数比`torch.log()`更准确。

如果输入是FloatTensor or DoubleTensor类型，则 `value` 必须为实数，否则须为整数。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(5)
>>> a

-0.4183
 0.3722
-0.3091
 0.4149
 0.5857
[torch.FloatTensor of size 5]

>>> torch.log1p(a)

-0.5418
 0.3164
-0.3697
 0.3471
 0.4611
[torch.FloatTensor of size 5]
```

torch.mul

```
torch.mul(input, value, out=None)
```

用标量值 `value` 乘以输入 `input` 的每个元素，并返回一个新的结果张量。 \(out=tensor ∗ value \)

如果输入是FloatTensor or DoubleTensor类型，则 `value` 必须为实数，否则须为整数。【译注：似乎并非如此，无关输入类型， `value` 取整数、实数皆可。】

参数：

- input (Tensor) – 输入张量
- value (Number) – 乘到每个元素的数
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(3)
>>> a

-0.9374
-0.5254
-0.6069
[torch.FloatTensor of size 3]

>>> torch.mul(a, 100)

-93.7411
-52.5374
-60.6908
[torch.FloatTensor of size 3]
```

```
torch.mul(input, other, out=None)
```

两个张量`input`,`other`按元素进行相乘，并返回到输出张量。即计算$out_i=input_i * other_i$

两计算张量形状不须匹配，但总元素数须一致。 注意：当形状不匹配时，`input`的形状作为输入张量的形状。

参数：

- input (Tensor) – 第一个相乘张量
- other (Tensor) – 第二个相乘张量
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4,4)
>>> a

-0.7280  0.0598 -1.4327 -0.5825
-0.1427 -0.0690  0.0821 -0.3270
-0.9241  0.5110  0.4070 -1.1188
-0.8308  0.7426 -0.6240 -1.1582
[torch.FloatTensor of size 4x4]

>>> b = torch.randn(2, 8)
>>> b

 0.0430 -1.0775  0.6015  1.1647 -0.6549  0.0308 -0.1670  1.0742
-1.2593  0.0292 -0.0849  0.4530  1.2404 -0.4659 -0.1840  0.5974
[torch.FloatTensor of size 2x8]

>>> torch.mul(a, b)

-0.0313 -0.0645 -0.8618 -0.6784
 0.0934 -0.0021 -0.0137 -0.3513
 1.1638  0.0149 -0.0346 -0.5068
-1.0304 -0.3460  0.1148 -0.6919
[torch.FloatTensor of size 4x4]
```

torch.neg

```
torch.neg(input, out=None) → Tensor
```

返回一个新张量，包含输入`input`张量按元素取负。 即，$out=-1*input$

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(5)
>>> a

-0.4430
 1.1690
-0.8836
-0.4565
 0.2968
[torch.FloatTensor of size 5]

>>> torch.neg(a)

 0.4430
-1.1690
 0.8836
 0.4565
-0.2968
[torch.FloatTensor of size 5]
```

torch.pow

```
torch.pow(input, exponent, out=None)
```

对输入 `input` 的按元素求 `exponent` 次幂值，并返回结果张量。 幂值 `exponent` 可以为单一 `float` 数或者与 `input` 相同元素数的张量。

当幂值为标量时，执行操作：

$$ out_i=x^{exponent} $$

当幂值为张量时，执行操作：

$$ out_i=x^{exponent_i} $$

参数：

- input (Tensor) – 输入张量
- exponent (float or Tensor) – 幂值
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

-0.5274
-0.8232
-2.1128
 1.7558
[torch.FloatTensor of size 4]

>>> torch.pow(a, 2)

 0.2781
 0.6776
 4.4640
 3.0829
[torch.FloatTensor of size 4]

>>> exp = torch.arange(1, 5)
>>> a = torch.arange(1, 5)
>>> a

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> exp

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> torch.pow(a, exp)

   1
   4
  27
 256
[torch.FloatTensor of size 4]
```

```
torch.pow(base, input, out=None)
```

`base` 为标量浮点值,`input` 为张量， 返回的输出张量 `out` 与输入张量相同形状。

执行操作为:

$$ out_i=base^{input_i} $$

参数：

- base (float) – 标量值，指数的底
- input (Tensor) – 幂值
- out (Tensor, optional) – 输出张量

例子：

```
>>> exp = torch.arange(1, 5)
>>> base = 2
>>> torch.pow(base, exp)

  2
  4
  8
 16
[torch.FloatTensor of size 4]
```

torch.reciprocal

```
torch.reciprocal(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的倒数，即 1.0/x。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.reciprocal(a)

 0.7210
 2.5565
-1.1583
-1.8289
[torch.FloatTensor of size 4]
```

torch.remainder

```
torch.remainder(input, divisor, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的除法余数。 除数与被除数可能同时包含整数或浮点数。余数与除数有相同的符号。

参数：

- input (Tensor) – 被除数
- divisor (Tensor or float) – 除数，一个数或者与除数相同大小的张量

- out (Tensor, optional) – 输出张量

例子：

```
>>> torch.remainder(torch.Tensor([-3, -2, -1, 1, 2, 3]), 2)
torch.FloatTensor([1, 0, 1, 1, 0, 1])
>>> torch.remainder(torch.Tensor([1, 2, 3, 4, 5]), 1.5)
torch.FloatTensor([1.0, 0.5, 0.0, 1.0, 0.5])
```

参考: 函数 `torch.fmod()` 同样可以计算除法余数，相当于 C 的 库函数 `fmod()`

torch.round

```
torch.round(input, out=None) → Tensor
```

返回一个新张量，将输入 `input` 张量每个元素舍入到最近的整数。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.2290
 1.3409
-0.5662
-0.0899
[torch.FloatTensor of size 4]

>>> torch.round(a)

 1
 1
-1
-0
[torch.FloatTensor of size 4]
```

torch.rsqrt

```
torch.rsqrt(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的平方根倒数。

参数：

- input (Tensor) – 输入张量

- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.2290
 1.3409
-0.5662
-0.0899
[torch.FloatTensor of size 4]

>>> torch.rsqrt(a)

 0.9020
 0.8636
    nan
    nan
[torch.FloatTensor of size 4]
```

torch.sigmoid

```
torch.sigmoid(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的sigmoid值。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

-0.4972
 1.3512
 0.1056
-0.2650
[torch.FloatTensor of size 4]

>>> torch.sigmoid(a)

 0.3782
 0.7943
 0.5264
 0.4341
[torch.FloatTensor of size 4]
```

torch.sign

```
torch.sign(input, out=None) → Tensor
```

符号函数：返回一个新张量，包含输入 `input` 张量每个元素的正负。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.sign(a)

-1
 1
 1
 1
[torch.FloatTensor of size 4]
```

torch.sin

```
torch.sin(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的正弦。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.sin(a)
-0.5944
 0.2684
 0.4322
 0.9667
[torch.FloatTensor of size 4]
```

torch.sinh

```
torch.sinh(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的双曲正弦。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.sinh(a)
-0.6804
 0.2751
 0.4619
 1.7225
[torch.FloatTensor of size 4]
```

torch.sqrt

```
torch.sqrt(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的平方根。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.2290
 1.3409
-0.5662
-0.0899
[torch.FloatTensor of size 4]

>>> torch.sqrt(a)

 1.1086
 1.1580
    nan
    nan
[torch.FloatTensor of size 4]
```

torch.tan

```
torch.tan(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的正切。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.tan(a)
-0.7392
 0.2786
 0.4792
 3.7801
[torch.FloatTensor of size 4]
```

torch.tanh

```
torch.tanh(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的双曲正切。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.tanh(a)
-0.5625
 0.2653
 0.4193
 0.8648
[torch.FloatTensor of size 4]
```

torch.trunc

```
torch.trunc(input, out=None) → Tensor
```

返回一个新张量，包含输入 `input` 张量每个元素的截断值(标量x的截断值是最接近其的整数，其比x更接近零。简而言之，有符号数的小数部分被舍弃)。

参数：

- input (Tensor) – 输入张量
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(4)
>>> a

-0.4972
 1.3512
 0.1056
-0.2650
[torch.FloatTensor of size 4]

>>> torch.trunc(a)

-0
 1
 0
-0
[torch.FloatTensor of size 4]
```

Reduction Ops

torch.cumprod

```
torch.cumprod(input, dim, out=None) → Tensor
```

返回输入沿指定维度的累积积。例如，如果输入是一个N 元向量，则结果也是一个N 元向量，第 `i` 个输出元素值为\(yi=x1∗x2∗x3∗...∗xi \)

参数：

- input (Tensor) – 输入张量
- dim (int) – 累积积操作的维度
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(10)
>>> a

 1.1148
 1.8423
 1.4143
-0.4403
 1.2859
-1.2514
-0.4748
 1.1735
-1.6332
-0.4272
[torch.FloatTensor of size 10]

>>> torch.cumprod(a, dim=0)

 1.1148
 2.0537
 2.9045
-1.2788
-1.6444
 2.0578
-0.9770
-1.1466
 1.8726
-0.8000
[torch.FloatTensor of size 10]

>>> a[5] = 0.0
>>> torch.cumprod(a, dim=0)

 1.1148
 2.0537
 2.9045
-1.2788
-1.6444
-0.0000
 0.0000
 0.0000
-0.0000
 0.0000
[torch.FloatTensor of size 10]
```

torch.cumsum

```
torch.cumsum(input, dim, out=None) → Tensor
```

返回输入沿指定维度的累积和。例如，如果输入是一个N元向量，则结果也是一个N元向量，第 `i` 个输出元素值为 \(yi=x1+x2+x3+...+xi\)

参数：

- input (Tensor) – 输入张量
- dim (int) – 累积和操作的维度
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(10)
>>> a

-0.6039
-0.2214
-0.3705
-0.0169
 1.3415
-0.1230
 0.9719
 0.6081
-0.1286
 1.0947
[torch.FloatTensor of size 10]

>>> torch.cumsum(a, dim=0)

-0.6039
-0.8253
-1.1958
-1.2127
 0.1288
 0.0058
 0.9777
 1.5858
 1.4572
 2.5519
[torch.FloatTensor of size 10]
```

torch.dist

```
torch.dist(input, other, p=2, out=None) → Tensor
```

返回 (`input` - `other`) 的 `p` 范数 。

参数：

- input (Tensor) – 输入张量

- other (Tensor) – 右侧输入张量
- p (float, optional) – 所计算的范数
- out (Tensor, optional) – 结果张量

例子：

```
>>> x = torch.randn(4)
>>> x

 0.2505
-0.4571
-0.3733
 0.7807
[torch.FloatTensor of size 4]

>>> y = torch.randn(4)
>>> y

 0.7782
-0.5185
 1.4106
-2.4063
[torch.FloatTensor of size 4]

>>> torch.dist(x, y, 3.5)
3.302832063224223
>>> torch.dist(x, y, 3)
3.3677282206393286
>>> torch.dist(x, y, 0)
inf
>>> torch.dist(x, y, 1)
5.560028076171875
```

torch.mean

```
torch.mean(input) → float
```

返回输入张量所有元素的均值。

参数： input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

-0.2946 -0.9143  2.1809
[torch.FloatTensor of size 1x3]

>>> torch.mean(a)
0.32398951053619385
```

```
torch.mean(input, dim, out=None) → Tensor
```

返回输入张量给定维度 `dim` 上每行的均值。

输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量
- dim (int) – the dimension to reduce
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4, 4)
>>> a

-1.2738 -0.3058  0.1230 -1.9615
 0.8771 -0.5430 -0.9233  0.9879
 1.4107  0.0317 -0.6823  0.2255
-1.3854  0.4953 -0.2160  0.2435
[torch.FloatTensor of size 4x4]

>>> torch.mean(a, 1)

-0.8545
 0.0997
 0.2464
-0.2157
[torch.FloatTensor of size 4x1]
```

torch.median

```
torch.median(input, dim=-1, values=None, indices=None) -> (Tensor, LongTensor)
```

返回输入张量给定维度每行的中位数，同时返回一个包含中位数的索引的 `LongTensor` 。

`dim` 值默认为输入张量的最后一维。 输出形状与输入相同，除了给定维度上为1.

注意: 这个函数还没有在 `torch.cuda.Tensor` 中定义

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- values (Tensor, optional) – 结果张量
- indices (Tensor, optional) – 返回的索引结果张量

```
>>> a

 -0.6891 -0.6662
 0.2697  0.7412
 0.5254 -0.7402
 0.5528 -0.2399
[torch.FloatTensor of size 4x2]

>>> a = torch.randn(4, 5)
>>> a

 0.4056 -0.3372  1.0973 -2.4884  0.4334
 2.1336  0.3841  0.1404 -0.1821 -0.7646
-0.2403  1.3975 -2.0068  0.1298  0.0212
-1.5371 -0.7257 -0.4871 -0.2359 -1.1724
[torch.FloatTensor of size 4x5]

>>> torch.median(a, 1)
(
 0.4056
 0.1404
 0.0212
-0.7257
[torch.FloatTensor of size 4x1]
,
 0
 2
 4
 1
[torch.LongTensor of size 4x1]
)
```

torch.mode

```
torch.mode(input, dim=-1, values=None, indices=None) -> (Tensor, LongTensor)
```

返回给定维 `dim` 上，每行的众数值。 同时返回一个 `LongTensor` ，包含众数职的索引。 `dim` 值默认为输入张量的最后一维。

输出形状与输入相同，除了给定维度上为1.

注意: 这个函数还没有在 `torch.cuda.Tensor` 中定义

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- values (Tensor, optional) – 结果张量
- indices (Tensor, optional) – 返回的索引张量

例子：

```
>>> a

 -0.6891 -0.6662
 0.2697  0.7412
 0.5254 -0.7402
 0.5528 -0.2399
[torch.FloatTensor of size 4x2]

>>> a = torch.randn(4, 5)
>>> a

 0.4056 -0.3372  1.0973 -2.4884  0.4334
 2.1336  0.3841  0.1404 -0.1821 -0.7646
-0.2403  1.3975 -2.0068  0.1298  0.0212
-1.5371 -0.7257 -0.4871 -0.2359 -1.1724
[torch.FloatTensor of size 4x5]

>>> torch.mode(a, 1)
(
-2.4884
-0.7646
-2.0068
-1.5371
[torch.FloatTensor of size 4x1]
,
 3
 4
 2
 0
[torch.LongTensor of size 4x1]
)
```

torch.norm

```
torch.norm(input, p=2) → float
```

返回输入张量 `input` 的p 范数。

参数：

- input (Tensor) – 输入张量
- p (float,optional) – 范数计算中的幂指数值

例子：

```
>>> a = torch.randn(1, 3)
>>> a

-0.4376 -0.5328  0.9547
[torch.FloatTensor of size 1x3]

>>> torch.norm(a, 3)
1.0338925067372466
```

```
torch.norm(input, p, dim, out=None) → Tensor
```

返回输入张量给定维 `dim` 上每行的p 范数。 输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量
- p (float) – 范数计算中的幂指数值
- dim (int) – 缩减的维度
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4, 2)
>>> a

-0.6891 -0.6662
 0.2697  0.7412
 0.5254 -0.7402
 0.5528 -0.2399
[torch.FloatTensor of size 4x2]

>>> torch.norm(a, 2, 1)

 0.9585
 0.7888
 0.9077
 0.6026
[torch.FloatTensor of size 4x1]

>>> torch.norm(a, 0, 1)

 2
 2
 2
 2
[torch.FloatTensor of size 4x1]
```

torch.prod

```
torch.prod(input) → float
```

返回输入张量 `input` 所有元素的积。

参数：input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

 0.6170  0.3546  0.0253
[torch.FloatTensor of size 1x3]

>>> torch.prod(a)
0.005537458061418483
```

```
torch.prod(input, dim, out=None) → Tensor
```

返回输入张量给定维度上每行的积。 输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4, 2)
>>> a

 0.1598 -0.6884
-0.1831 -0.4412
-0.9925 -0.6244
-0.2416 -0.8080
[torch.FloatTensor of size 4x2]

>>> torch.prod(a, 1)

-0.1100
 0.0808
 0.6197
 0.1952
[torch.FloatTensor of size 4x1]
```

torch.std

```
torch.std(input) → float
```

返回输入张量 `input` 所有元素的标准差。

参数：- input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

-1.3063  1.4182 -0.3061
[torch.FloatTensor of size 1x3]

>>> torch.std(a)
1.3782334731508061
```

```
torch.std(input, dim, out=None) → Tensor
```

返回输入张量给定维度上每行的标准差。 输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4, 4)
>>> a

 0.1889 -2.4856  0.0043  1.8169
-0.7701 -0.4682 -2.2410  0.4098
 0.1919 -1.1856 -1.0361  0.9085
 0.0173  1.0662  0.2143 -0.5576
[torch.FloatTensor of size 4x4]

>>> torch.std(a, dim=1)

 1.7756
 1.1025
 1.0045
 0.6725
[torch.FloatTensor of size 4x1]
```

torch.sum

```
torch.sum(input) → float
```

返回输入张量 `input` 所有元素的和。

输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

 0.6170  0.3546  0.0253
[torch.FloatTensor of size 1x3]

>>> torch.sum(a)
0.9969287421554327
```

```
torch.sum(input, dim, out=None) → Tensor
```

返回输入张量给定维度上每行的和。 输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4, 4)
>>> a

-0.4640  0.0609  0.1122  0.4784
-1.3063  1.6443  0.4714 -0.7396
-1.3561 -0.1959  1.0609 -1.9855
 2.6833  0.5746 -0.5709 -0.4430
[torch.FloatTensor of size 4x4]

>>> torch.sum(a, 1)

 0.1874
 0.0698
-2.4767
 2.2440
[torch.FloatTensor of size 4x1]
```

torch.var

```
torch.var(input) → float
```

返回输入张量所有元素的方差

输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

-1.3063  1.4182 -0.3061
[torch.FloatTensor of size 1x3]

>>> torch.var(a)
1.899527506513334
```

```
torch.var(input, dim, out=None) → Tensor
```

返回输入张量给定维度上每行的方差。 输出形状与输入相同，除了给定维度上为1.

参数：

- input (Tensor) – 输入张量
- dim (int) – the dimension to reduce
- out (Tensor, optional) – 结果张量 例子： ```python

> > > a = torch.randn(4, 4) a

-1.2738 -0.3058 0.1230 -1.9615 0.8771 -0.5430 -0.9233 0.9879 1.4107 0.0317 -0.6823 0.2255 -1.3854 0.4953 -0.2160 0.2435 [torch.FloatTensor of size 4x4]

> > > torch.var(a, 1)

0.8859 0.9509 0.7548 0.6949 [torch.FloatTensor of size 4x1]

```
## 比较操作 Comparison Ops

### torch.eq
```python
torch.eq(input, other, out=None) → Tensor
```

比较元素相等性。第二个参数可为一个数或与第一个参数同类型形状的张量。

参数：

- input (Tensor) – 待比较张量
- other (Tensor or float) – 比较张量或数
- out (Tensor, optional) – 输出张量，须为 ByteTensor类型 or 与 `input` 同类型

返回值： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果(相等为1，不等为0)

返回类型： Tensor

例子：

```
>>> torch.eq(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
1  0
0  1
[torch.ByteTensor of size 2x2]
```

torch.equal

```
torch.equal(tensor1, tensor2) → bool
```

如果两个张量有相同的形状和元素值，则返回 `True` ，否则 `False` 。

例子：
```

```
>>> torch.equal(torch.Tensor([1, 2]), torch.Tensor([1, 2]))
True
```

torch.ge

```
torch.ge(input, other, out=None) → Tensor
```

逐元素比较 `input` 和 `other` ，即是否 \(input >= other \)。

如果两个张量有相同的形状和元素值，则返回 `True` ，否则 `False` 。 第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

- input (Tensor) – 待对比的张量
- other (Tensor or float) – 对比的张量或 `float` 值
- out (Tensor, optional) – 输出张量。必须为 `ByteTensor` 或者与第一个参数 `tensor` 相同类型。

返回值： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果(是否 input >= other)。
返回类型： Tensor

例子：

```
>>> torch.ge(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 1  1
 0  1
[torch.ByteTensor of size 2x2]
```

torch.gt

```
torch.gt(input, other, out=None) → Tensor
```

逐元素比较 `input` 和 `other` ， 即是否\(input > other \) 如果两个张量有相同的形状和元素值，则返回 `True` ，否则 `False` 。 第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

- input (Tensor) – 要对比的张量
- other (Tensor or float) – 要对比的张量或 `float` 值
- out (Tensor, optional) – 输出张量。必须为 `ByteTensor` 或者与第一个参数 `tensor` 相同类型。

返回值： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果(是否 input >= other)。
返回类型： Tensor

例子：

```
>>> torch.gt(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  1
 0  0
[torch.ByteTensor of size 2x2]
```

torch.kthvalue

```
torch.kthvalue(input, k, dim=None, out=None) -> (Tensor, LongTensor)
```

取输入张量 `input` 指定维上第k 个最小值。如果不指定 `dim` ，则默认为 `input` 的最后一维。

返回一个元组 *(values,indices)*，其中 `indices` 是原始输入张量 `input` 中沿 `dim` 维的第 `k` 个最小值下标。

参数:

- input (Tensor) – 要对比的张量
- k (int) – 第 `k` 个最小值
- dim (int, optional) – 沿着此维进行排序
- out (tuple, optional) – 输出元组 (Tensor, LongTensor) 可选地给定作为 输出 buffers

例子：

```
>>> x = torch.arange(1, 6)
>>> x

 1
 2
 3
 4
 5
[torch.FloatTensor of size 5]

>>> torch.kthvalue(x, 4)
(
 4
[torch.FloatTensor of size 1]
,
 3
[torch.LongTensor of size 1]
)
```

torch.le

```
torch.le(input, other, out=None) → Tensor
```

逐元素比较 `input` 和 `other` ，即是否\(input <= other \) 第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

- input (Tensor) – 要对比的张量
- other (Tensor or float) – 对比的张量或 `float` 值
- out (Tensor, optional) – 输出张量。必须为 `ByteTensor` 或者与第一个参数 `tensor` 相同类型。

返回值： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果(是否 input >= other)。
返回类型： Tensor

例子：

```
>>> torch.le(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 1  0
 1  1
[torch.ByteTensor of size 2x2]
```

torch.lt

```
torch.lt(input, other, out=None) → Tensor
```

逐元素比较 `input` 和 `other` ，即是否 \(input < other \)

第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

- input (Tensor) – 要对比的张量
- other (Tensor or float) – 对比的张量或 `float` 值
- out (Tensor, optional) – 输出张量。必须为 `ByteTensor` 或者与第一个参数 `tensor` 相同类型。

input： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果(是否 tensor >= other)。
返回类型： Tensor

例子：

```
>>> torch.lt(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  0
 1  0
[torch.ByteTensor of size 2x2]
```

torch.max

```
torch.max()
```

返回输入张量所有元素的最大值。

参数:

- input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

 0.4729 -0.2266 -0.2085
[torch.FloatTensor of size 1x3]

>>> torch.max(a)
0.4729
```

```
torch.max(input, dim, max=None, max_indices=None) -> (Tensor, LongTensor)
```

返回输入张量给定维度上每行的最大值，并同时返回每个最大值的位置索引。

输出形状中，将 `dim` 维设定为1，其它与输入形状保持一致。

参数:

- input (Tensor) – 输入张量
- dim (int) – 指定的维度
- max (Tensor, optional) – 结果张量，包含给定维度上的最大值
- max_indices (LongTensor, optional) – 结果张量，包含给定维度上每个最大值的位置索引

例子：

```
>> a = torch.randn(4, 4)
>> a

0.0692  0.3142  1.2513 -0.5428
0.9288  0.8552 -0.2073  0.6409
1.0695 -0.0101 -2.4507 -1.2230
0.7426 -0.7666  0.4862 -0.6628
torch.FloatTensor of size 4x4]

>>> torch.max(a, 1)
(
 1.2513
 0.9288
 1.0695
 0.7426
[torch.FloatTensor of size 4x1]
,
 2
 0
 0
 0
[torch.LongTensor of size 4x1]
)
```

```
torch.max(input, other, out=None) → Tensor
```

返回输入张量给定维度上每行的最大值，并同时返回每个最大值的位置索引。 即，\(out_i=max(input_i,other_i) \)

输出形状中，将`dim`维设定为1，其它与输入形状保持一致。

参数:

- input (Tensor) – 输入张量
- other (Tensor) – 输出张量
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> b = torch.randn(4)
>>> b

 1.0067
-0.8010
 0.6258
 0.3627
[torch.FloatTensor of size 4]

>>> torch.max(a, b)

 1.3869
 0.3912
 0.6258
 0.3627
[torch.FloatTensor of size 4]
```

torch.min

```
torch.min(input) → float
```

返回输入张量所有元素的最小值。

参数: input (Tensor) – 输入张量

例子：

```
>>> a = torch.randn(1, 3)
>>> a

 0.4729 -0.2266 -0.2085
[torch.FloatTensor of size 1x3]

>>> torch.min(a)
-0.22663167119026184
```

```
torch.min(input, dim, min=None, min_indices=None) -> (Tensor, LongTensor)
```

返回输入张量给定维度上每行的最小值，并同时返回每个最小值的位置索引。

输出形状中，将 `dim` 维设定为1，其它与输入形状保持一致。

参数:

- input (Tensor) – 输入张量
- dim (int) – 指定的维度

- min (Tensor, optional) – 结果张量，包含给定维度上的最小值
- min_indices (LongTensor, optional) – 结果张量，包含给定维度上每个最小值的位置索引

例子：

```
>> a = torch.randn(4, 4)
>> a

0.0692  0.3142  1.2513 -0.5428
0.9288  0.8552 -0.2073  0.6409
1.0695 -0.0101 -2.4507 -1.2230
0.7426 -0.7666  0.4862 -0.6628
torch.FloatTensor of size 4x4]

>> torch.min(a, 1)

0.5428
0.2073
2.4507
0.7666
torch.FloatTensor of size 4x1]

3
2
2
1
torch.LongTensor of size 4x1]
```

```
torch.min(input, other, out=None) → Tensor
```

`input` 中逐元素与 `other` 相应位置的元素对比，返回最小值到输出张量。即，\(out_i = min(tensor_i, other_i)\)

两张量形状不需匹配，但元素数须相同。

注意：当形状不匹配时，`input` 的形状作为返回张量的形状。

参数:

- input (Tensor) – 输入张量
- other (Tensor) – 第二个输入张量
- out (Tensor, optional) – 结果张量

例子：

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> b = torch.randn(4)
>>> b

 1.0067
-0.8010
 0.6258
 0.3627
[torch.FloatTensor of size 4]

>>> torch.min(a, b)

 1.0067
-0.8010
-0.8634
-0.5468
[torch.FloatTensor of size 4]
```

torch.ne

```
torch.ne(input, other, out=None) → Tensor
```

逐元素比较 `input` 和 `other` ，即是否 \(input != other \)。第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

- input (Tensor) – 待对比的张量
- other (Tensor or float) – 对比的张量或 `float` 值
- out (Tensor, optional) – 输出张量。必须为 `ByteTensor` 或者与 `input` 相同类型。

返回值： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果 (如果 tensor != other 为 `True` ，返回 `1`)。

返回类型： Tensor

例子：

```
>>> torch.ne(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  1
 1  0
[torch.ByteTensor of size 2x2]
```

torch.sort

```
torch.sort(input, dim=None, descending=False, out=None) -> (Tensor, LongTensor)
```

对输入张量 `input` 沿着指定维按升序排序。如果不给定 `dim` ，则默认为输入的最后一维。如果指定参数 `descending` 为 `True` ，则按降序排序

返回元组 (sorted_tensor, sorted_indices) ， `sorted_indices` 为原始输入中的下标。

参数:

- input (Tensor) – 要对比的张量
- dim (int, optional) – 沿着此维排序
- descending (bool, optional) – 布尔值，控制升降排序
- out (tuple, optional) – 输出张量。必须为 `ByteTensor` 或者与第一个参数 `tensor` 相同类型。

例子：

```
>>> x = torch.randn(3, 4)
>>> sorted, indices = torch.sort(x)
>>> sorted

-1.6747  0.0610  0.1190  1.4137
-1.4782  0.7159  1.0341  1.3678
-0.3324 -0.0782  0.3518  0.4763
[torch.FloatTensor of size 3x4]

>>> indices

 0  1  3  2
 2  1  0  3
 3  1  0  2
[torch.LongTensor of size 3x4]

>>> sorted, indices = torch.sort(x, 0)
>>> sorted

-1.6747 -0.0782 -1.4782 -0.3324
 0.3518  0.0610  0.4763  0.1190
 1.0341  0.7159  1.4137  1.3678
[torch.FloatTensor of size 3x4]

>>> indices

 0  2  1  2
 2  0  2  0
 1  1  0  1
[torch.LongTensor of size 3x4]
```

torch.topk

```
torch.topk(input, k, dim=None, largest=True, sorted=True, out=None) -> (Tensor, LongTe
nsor)
```

沿给定 `dim` 维度返回输入张量 `input` 中 `k` 个最大值。 如果不指定 `dim` ，则默认为 `input` 的最后一维。 如果为 `largest` 为 `False` ，则返回最小的 `k` 个值。

返回一个元组 *(values,indices)*，其中 `indices` 是原始输入张量 `input` 中测元素下标。 如果设定布尔值 `sorted` 为 *True*，将会确保返回的 `k` 个值被排序。

参数:

- input (Tensor) – 输入张量
- k (int) – “top-k”中的 `k`
- dim (int, optional) – 排序的维
- largest (bool, optional) – 布尔值，控制返回最大或最小值
- sorted (bool, optional) – 布尔值，控制返回值是否排序
- out (tuple, optional) – 可选输出张量 (Tensor, LongTensor) output buffers

```
>>> x = torch.arange(1, 6)
>>> x

 1
 2
 3
 4
 5
[torch.FloatTensor of size 5]

>>> torch.topk(x, 3)
(
 5
 4
 3
[torch.FloatTensor of size 3]
,
 4
 3
 2
[torch.LongTensor of size 3]
)
>>> torch.topk(x, 3, 0, largest=False)
(
 1
 2
 3
[torch.FloatTensor of size 3]
,
 0
 1
 2
[torch.LongTensor of size 3]
)
```

其它操作 Other Operations

torch.cross

```
torch.cross(input, other, dim=-1, out=None) → Tensor
```

返回沿着维度 `dim` 上，两个张量 `input` 和 `other` 的向量积（叉积）。 `input` 和 `other` 必须有相同的形状，且指定的 `dim` 维上size必须为 `3` 。

如果不指定 `dim` ，则默认为第一个尺度为 `3` 的维。

参数：

- input (Tensor) – 输入张量
- other (Tensor) – 第二个输入张量
- dim (int, optional) – 沿着此维进行叉积操作
- out (Tensor,optional) – 结果张量

例子：

```
>>> a = torch.randn(4, 3)
>>> a

-0.6652 -1.0116 -0.6857
 0.2286  0.4446 -0.5272
 0.0476  0.2321  1.9991
 0.6199  1.1924 -0.9397
[torch.FloatTensor of size 4x3]

>>> b = torch.randn(4, 3)
>>> b

-0.1042 -1.1156  0.1947
 0.9947  0.1149  0.4701
-1.0108  0.8319 -0.0750
 0.9045 -1.3754  1.0976
[torch.FloatTensor of size 4x3]

>>> torch.cross(a, b, dim=1)

-0.9619  0.2009  0.6367
 0.2696 -0.6318 -0.4160
-1.6805 -2.0171  0.2741
 0.0163 -1.5304 -1.9311
[torch.FloatTensor of size 4x3]

>>> torch.cross(a, b)

-0.9619  0.2009  0.6367
 0.2696 -0.6318 -0.4160
-1.6805 -2.0171  0.2741
 0.0163 -1.5304 -1.9311
[torch.FloatTensor of size 4x3]
```

torch.diag

```
torch.diag(input, diagonal=0, out=None) → Tensor
```

- 如果输入是一个向量(1D 张量)，则返回一个以 `input` 为对角线元素的2D方阵
- 如果输入是一个矩阵(2D 张量)，则返回一个包含 `input` 对角线元素的1D张量

参数 `diagonal` 指定对角线:

- `diagonal` = 0, 主对角线
- `diagonal` > 0, 主对角线之上
- `diagonal` < 0, 主对角线之下

参数：

- input (Tensor) – 输入张量
- diagonal (int, optional) – 指定对角线
- out (Tensor, optional) – 输出张量

例子：

- 取得以 `input` 为对角线的方阵： ```python

    > > > a = torch.randn(3) a

    1.0480 -2.3405 -1.1138 [torch.FloatTensor of size 3]

> > > torch.diag(a)

1.0480 0.0000 0.0000 0.0000 -2.3405 0.0000 0.0000 0.0000 -1.1138 [torch.FloatTensor of size 3x3]

> > > torch.diag(a, 1)

0.0000 1.0480 0.0000 0.0000 0.0000 0.0000 -2.3405 0.0000 0.0000 0.0000 0.0000 -1.1138 0.0000 0.0000 0.0000 0.0000 [torch.FloatTensor of size 4x4]

```
- 取得给定矩阵第`k`个对角线:
```

> > > a = torch.randn(3, 3) a

-1.5328 -1.3210 -1.5204 0.8596 0.0471 -0.2239 -0.6617 0.0146 -1.0817 [torch.FloatTensor of size 3x3]

> > > torch.diag(a, 0)

-1.5328 0.0471 -1.0817 [torch.FloatTensor of size 3]

> > > torch.diag(a, 1)

-1.3210 -0.2239 [torch.FloatTensor of size 2]

```
### torch.histc
```python
torch.histc(input, bins=100, min=0, max=0, out=None) → Tensor
```
```

计算输入张量的直方图。以 `min` 和 `max` 为range边界，将其均分成 `bins` 个直条，然后将排序好的数据划分到各个直条(bins)中。如果 `min` 和 `max` 都为0, 则利用数据中的最大最小值作为边界。

参数：

- input (Tensor) – 输入张量
- bins (int) – 直方图 bins(直条)的个数(默认100个)
- min (int) – range的下边界(包含)
- max (int) – range的上边界(包含)
- out (Tensor, optional) – 结果张量

返回： 直方图 返回类型：张量

例子：

```
>>> torch.histc(torch.FloatTensor([1, 2, 1]), bins=4, min=0, max=3)
FloatTensor([0, 2, 1, 0])
```

torch.renorm

```
torch.renorm(input, p, dim, maxnorm, out=None) → Tensor
```

返回一个张量，包含规范化后的各个子张量，使得沿着 `dim` 维划分的各子张量的p范数小于 `maxnorm` 。

注意 如果p范数的值小于 `maxnorm` ，则当前子张量不需要修改。

注意: 更详细解释参考torch7 以及Hinton et al. 2012, p. 2

参数：

- input (Tensor) – 输入张量
- p (float) – 范数的p
- dim (int) – 沿着此维切片，得到张量子集
- maxnorm (float) – 每个子张量的范数的最大值
- out (Tensor, optional) – 结果张量

例子：

```
>>> x = torch.ones(3, 3)
>>> x[1].fill_(2)
>>> x[2].fill_(3)
>>> x

 1  1  1
 2  2  2
 3  3  3
[torch.FloatTensor of size 3x3]

>>> torch.renorm(x, 1, 0, 5)

 1.0000  1.0000  1.0000
 1.6667  1.6667  1.6667
 1.6667  1.6667  1.6667
[torch.FloatTensor of size 3x3]
```

torch.trace

```
torch.trace(input) → float
```

返回输入2维矩阵对角线元素的和(迹)

例子：

```
>>> x = torch.arange(1, 10).view(3, 3)
>>> x

 1  2  3
 4  5  6
 7  8  9
[torch.FloatTensor of size 3x3]

>>> torch.trace(x)
15.0
```

torch.tril

```
torch.tril(input, k=0, out=None) → Tensor
```

返回一个张量 `out` ，包含输入矩阵(2D张量)的下三角部分， `out` 其余部分被设为 `0` 。这里所说的下三角部分为矩阵指定对角线 `diagonal` 之上的元素。

参数 `k` 控制对角线:

- `k` = 0, 主对角线
- `k` > 0, 主对角线之上
- `k` < 0, 主对角线之下

参数：

- input (Tensor) – 输入张量

- k (int, optional) – 指定对角线
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(3,3)
>>> a

 1.3225  1.7304  1.4573
-0.3052 -0.3111 -0.1809
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.tril(a)

 1.3225  0.0000  0.0000
-0.3052 -0.3111  0.0000
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.tril(a, k=1)

 1.3225  1.7304  0.0000
-0.3052 -0.3111 -0.1809
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.tril(a, k=-1)

 0.0000  0.0000  0.0000
-0.3052  0.0000  0.0000
 1.2469  0.0064  0.0000
[torch.FloatTensor of size 3x3]
```

torch.triu

```
torch.triu(input, k=0, out=None) → Tensor
```

返回一个张量，包含输入矩阵(2D张量)的上三角部分，其余部分被设为 `0` 。这里所说的上三角部分为矩阵指定对角线 `diagonal` 之上的元素。

参数 `k` 控制对角线:

- `k` = 0, 主对角线
- `k` > 0, 主对角线之上
- `k` < 0, 主对角线之下

参数：

- input (Tensor) – 输入张量
- k (int, optional) – 指定对角线
- out (Tensor, optional) – 输出张量

例子：

```
>>> a = torch.randn(3,3)
>>> a

 1.3225  1.7304  1.4573
-0.3052 -0.3111 -0.1809
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.triu(a)

 1.3225  1.7304  1.4573
 0.0000 -0.3111 -0.1809
 0.0000  0.0000 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.triu(a, k=1)

 0.0000  1.7304  1.4573
 0.0000  0.0000 -0.1809
 0.0000  0.0000  0.0000
[torch.FloatTensor of size 3x3]

>>> torch.triu(a, k=-1)

 1.3225  1.7304  1.4573
-0.3052 -0.3111 -0.1809
 0.0000  0.0064 -1.6250
[torch.FloatTensor of size 3x3]
```

BLAS and LAPACK Operations

torch.addbmm

```
torch.addbmm(beta=1, mat, alpha=1, batch1, batch2, out=None) → Tensor
```

对两个批 `batch1` 和 `batch2` 内存储的矩阵进行批矩阵乘操作，附带reduced add 步骤(所有矩阵乘结果沿着第一维相加)。矩阵 `mat` 加到最终结果。 `batch1` 和 `batch2` 都为包含相同数量矩阵的3维张量。 如果 `batch1` 是形为\(b\times n \times m \)的张量， `batch1` 是形为\(b\times m \times p \)的张量，则 `out` 和 `mat` 的形状都是\(n \times p \)，即 \(res=(beta∗M)+(alpha∗sum(batch1_i@batch2_i,i=0,b)) \)

对类型为 *FloatTensor* 或 *DoubleTensor* 的输入， `alpha` and `beta` 必须为实数，否则两个参数须为整数。

参数：

- beta (Number, optional) – 用于 `mat` 的乘子
- mat (Tensor) – 相加矩阵
- alpha (Number, optional) – 用于\(batch1 @ batch2 \)的乘子
- batch1 (Tensor) – 第一批相乘矩阵
- batch2 (Tensor) – 第二批相乘矩阵
- out (Tensor, optional) – 输出张量

例子:

```
>>> M = torch.randn(3, 5)
>>> batch1 = torch.randn(10, 3, 4)
>>> batch2 = torch.randn(10, 4, 5)
>>> torch.addbmm(M, batch1, batch2)

 -3.1162  11.0071   7.3102   0.1824  -7.6892
  1.8265   6.0739   0.4589  -0.5641  -5.4283
 -9.3387  -0.1794  -1.2318  -6.8841  -4.7239
[torch.FloatTensor of size 3x5]
```

torch.addmm

```
torch.addmm(beta=1, mat, alpha=1, mat1, mat2, out=None) → Tensor
```

对矩阵 `mat1` 和 `mat2` 进行矩阵乘操作。矩阵 `mat` 加到最终结果。如果 `mat1` 是一个 \(n \times m \)张量， `mat2` 是一个 \(m \times p \)张量，那么 `out` 和 `mat` 的形状为\(n \times p \)。 *alpha* 和 *beta* 分别是两个矩阵 \(mat1 @ mat2 \)和\(mat \)的比例因子，即， \(out=(beta∗M)+(alpha∗mat1@mat2) \)

对类型为 *FloatTensor* 或 *DoubleTensor* 的输入， `beta` and `alpha` 必须为实数，否则两个参数须为整数。

参数 ：

- beta (Number, optional) – 用于 `mat` 的乘子
- mat (Tensor) – 相加矩阵
- alpha (Number, optional) – 用于\(mat1 @ mat2 \)的乘子
- mat1 (Tensor) – 第一个相乘矩阵
- mat2 (Tensor) – 第二个相乘矩阵
- out (Tensor, optional) – 输出张量

```
>>> M = torch.randn(2, 3)
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.addmm(M, mat1, mat2)

-0.4095 -1.9703  1.3561
 5.7674 -4.9760  2.7378
[torch.FloatTensor of size 2x3]
```

torch.addmv

```
torch.addmv(beta=1, tensor, alpha=1, mat, vec, out=None) → Tensor
```

对矩阵 `mat` 和向量 `vec` 对进行相乘操作。向量 `tensor` 加到最终结果。如果 `mat` 是一个 \(n \times m \)维矩阵， `vec` 是一个 \(m \)维向量，那么 `out` 和 `mat` 的为\(n \)元向量。 可选参数*alpha* 和 *beta* 分别是 \(mat * vec \)和\(mat \)的比例因子，即， \(out=(beta∗tensor)+(alpha∗(mat@vec)) \)

对类型为*FloatTensor*或*DoubleTensor*的输入， `alpha` and `beta` 必须为实数，否则两个参数须为整数。

参数 ：

- beta (Number, optional) – 用于 `mat` 的乘子
- mat (Tensor) – 相加矩阵
- alpha (Number, optional) – 用于\(mat1 @ vec \)的乘子
- mat (Tensor) – 相乘矩阵
- vec (Tensor) – 相乘向量
- out (Tensor, optional) – 输出张量

```
>>> M = torch.randn(2)
>>> mat = torch.randn(2, 3)
>>> vec = torch.randn(3)
>>> torch.addmv(M, mat, vec)

-2.0939
-2.2950
[torch.FloatTensor of size 2]
```

torch.addr

```
torch.addr(beta=1, mat, alpha=1, vec1, vec2, out=None) → Tensor
```

对向量 `vec1` 和 `vec2` 对进行张量积操作。矩阵 `mat` 加到最终结果。如果 `vec1` 是一个 \(n \)维向量， `vec2` 是一个 \(m \)维向量，那么矩阵 `mat` 的形状须为\(n \times m \)。 可选参数*beta* 和 *alpha* 分别是两个矩阵 \(mat \)和 \(vec1 @ vec2 \)的比例因子，即，\(resi=(beta∗Mi)+(alpha∗batch1i×batch2i)\)

对类型为*FloatTensor*或*DoubleTensor*的输入， `alpha` and `beta` 必须为实数，否则两个参数须为整数。

参数 ：

- beta (Number, optional) – 用于 `mat` 的乘子
- mat (Tensor) – 相加矩阵
- alpha (Number, optional) – 用于两向量\(vec1， vec2 \)外积的乘子
- vec1 (Tensor) – 第一个相乘向量
- vec2 (Tensor) – 第二个相乘向量
- out (Tensor, optional) – 输出张量

```
>>> vec1 = torch.arange(1, 4)
>>> vec2 = torch.arange(1, 3)
>>> M = torch.zeros(3, 2)
>>> torch.addr(M, vec1, vec2)
 1  2
 2  4
 3  6
[torch.FloatTensor of size 3x2]
```

torch.baddbmm

```
torch.baddbmm(beta=1, mat, alpha=1, batch1, batch2, out=None) → Tensor
```

对两个批 `batch1` 和 `batch2` 内存储的矩阵进行批矩阵乘操作，矩阵 `mat` 加到最终结果。 `batch1` 和 `batch2` 都为包含相同数量矩阵的3维张量。 如果 `batch1` 是形为\(b\times n \times m \)的张量， `batch1` 是形为\(b\times m \times p \)的张量，则 `out` 和 `mat` 的形状都是\(n \times p \)，即 \(resi=(beta∗M_i)+(alpha∗batch1_i×batch2_i) \)

对类型为*FloatTensor*或*DoubleTensor*的输入， `alpha` and `beta` 必须为实数，否则两个参数须为整数。

参数：

- beta (Number, optional) – 用于 `mat` 的乘子
- mat (Tensor) – 相加矩阵
- alpha (Number, optional) – 用于\(batch1 @ batch2 \)的乘子
- batch1 (Tensor) – 第一批相乘矩阵
- batch2 (Tensor) – 第二批相乘矩阵
- out (Tensor, optional) – 输出张量

```
>>> M = torch.randn(10, 3, 5)
>>> batch1 = torch.randn(10, 3, 4)
>>> batch2 = torch.randn(10, 4, 5)
>>> torch.baddbmm(M, batch1, batch2).size()
torch.Size([10, 3, 5])
```

torch.bmm

```
torch.bmm(batch1, batch2, out=None) → Tensor
```

对存储在两个批 `batch1` 和 `batch2` 内的矩阵进行批矩阵乘操作。 `batch1` 和 `batch2` 都为包含相同数量矩阵的3维张量。 如果 `batch1` 是形为\(b\times n \times m \)的张量， `batch1` 是形为\(b\times m \times p \)的张量，则 `out` 和 `mat` 的形状都是\(n \times p \)，即 \(res=(beta∗M)+(alpha∗sum(batch1_i@batch2_i,i=0,b)) \)

对类型为 *FloatTensor* 或 *DoubleTensor* 的输入，`alpha` and `beta` 必须为实数，否则两个参数须为整数。

参数：

- batch1 (Tensor) – 第一批相乘矩阵
- batch2 (Tensor) – 第二批相乘矩阵
- out (Tensor, optional) – 输出张量

```
>>> batch1 = torch.randn(10, 3, 4)
>>> batch2 = torch.randn(10, 4, 5)
>>> res = torch.bmm(batch1, batch2)
>>> res.size()
torch.Size([10, 3, 5])
```

torch.btrifact

```
torch.btrifact(A, info=None) → Tensor, IntTensor
```

返回一个元组，包含LU 分解和 `pivots` 。可选参数 `info` 决定是否对每个minibatch样本进行分解。`info` are from dgetrf and a non-zero value indicates an error occurred. 如果用CUDA的话，这个值来自于CUBLAS，否则来自LAPACK。

参数： A (Tensor) – 待分解张量

```
>>> A = torch.randn(2, 3, 3)
>>> A_LU = A.btrifact()
```

torch.btrisolve

```
torch.btrisolve(b, LU_data, LU_pivots) → Tensor
```

返回线性方程组\(Ax = b \)的LU解。

参数：

- b (Tensor) – RHS 张量.
- LU_data (Tensor) – Pivoted LU factorization of A from btrifact.
- LU_pivots (IntTensor) – LU 分解的Pivots.

例子：

```
>>> A = torch.randn(2, 3, 3)
>>> b = torch.randn(2, 3)
>>> A_LU = torch.btrifact(A)
>>> x = b.btrisolve(*A_LU)
>>> torch.norm(A.bmm(x.unsqueeze(2)) - b)
6.664001874625056e-08
```

torch.dot

```
torch.dot(tensor1, tensor2) → float
```

计算两个张量的点乘(内乘),两个张量都为1-D 向量.

例子：

```
>>> torch.dot(torch.Tensor([2, 3]), torch.Tensor([2, 1]))
7.0
```

torch.eig

```
torch.eig(a, eigenvectors=False, out=None) -> (Tensor, Tensor)
```

计算实方阵 `a` 的特征值和特征向量

参数：

- a (Tensor) – 方阵，待计算其特征值和特征向量
- eigenvectors (bool) – 布尔值，如果 `True` ，则同时计算特征值和特征向量，否则只计算特征值。
- out (tuple, optional) – 输出元组

返回值： 元组，包括：

- e (Tensor): a 的右特征向量
- v (Tensor): 如果 `eigenvectors` 为 `True` ，则为包含特征向量的张量; 否则为空张量

返回值类型： (Tensor, Tensor)

torch.gels

```
torch.gels(B, A, out=None) → Tensor
```

对形如\(m \times n \)的满秩矩阵 `a` 计算其最小二乘和最小范数问题的解。 如果\(m >= n \), `gels` 对最小二乘问题进行求解，即：

$$minimize \qquad ‖AX - B‖_F $$

如果\(m < n \), `gels` 求解最小范数问题，即：

$$minimize \qquad ‖ X ‖_F \qquad subject \ to \quad a \quad b AX=B $$

返回矩阵\(X \)的前\(n \) 行包含解。余下的行包含以下残差信息: 相应列从第n 行开始计算的每列的欧式距离。

注意： 返回矩阵总是被转置，无论输入矩阵的原始布局如何，总会被转置；即，总是有 stride (1, m) 而不是 (m, 1).

参数：

- B (Tensor) – 矩阵B
- A (Tensor) – \(m \times n \)矩阵
- out (tuple, optional) – 输出元组

返回值： 元组，包括：

- X (Tensor): 最小二乘解
- qr (Tensor): QR 分解的细节

返回值类型： (Tensor, Tensor)

例子：

```
>>> A = torch.Tensor([[1, 1, 1],
...                   [2, 3, 4],
...                   [3, 5, 2],
...                   [4, 2, 5],
...                   [5, 4, 3]])
>>> B = torch.Tensor([[-10, -3],
                      [ 12, 14],
                      [ 14, 12],
                      [ 16, 16],
                      [ 18, 16]])
>>> X, _ = torch.gels(B, A)
>>> X
2.0000  1.0000
1.0000  1.0000
1.0000  2.0000
[torch.FloatTensor of size 3x2]
```

torch.geqrf

```
torch.geqrf(input, out=None) -> (Tensor, Tensor)
```

这是一个直接调用LAPACK的底层函数。 一般使用 `torch.qr()`

计算输入的QR 分解，但是并不会分别创建Q,R两个矩阵，而是直接调用LAPACK 函数 Rather, this directly calls the underlying LAPACK function ?geqrf which produces a sequence of 'elementary reflectors'.

参考 LAPACK文档获取更详细信息。

参数:

- input (Tensor) – 输入矩阵
- out (tuple, optional) – 元组，包含输出张量 (Tensor, Tensor)

torch.ger

```
torch.ger(vec1, vec2, out=None) → Tensor
```

计算两向量 `vec1`, `vec2` 的张量积。如果 `vec1` 的长度为 `n`, `vec2` 长度为 `m` ，则输出 `out` 应为形如n x m的矩阵。

参数:

- vec1 (Tensor) – 1D 输入向量
- vec2 (Tensor) – 1D 输入向量
- out (tuple, optional) – 输出张量

例子：

```
>>> v1 = torch.arange(1, 5)
>>> v2 = torch.arange(1, 4)
>>> torch.ger(v1, v2)

  1   2   3
  2   4   6
  3   6   9
  4   8  12
[torch.FloatTensor of size 4x3]
```

torch.gesv

```
torch.gesv(B, A, out=None) -> (Tensor, Tensor)
```

\(X, LU = torch.gesv(B, A) \)，返回线性方程组\(AX=B \)的解。

LU 包含两个矩阵L，U。A须为非奇异方阵，如果A是一个\(m \times m \)矩阵，B 是\(m \times k \)矩阵，则LU 是\(m \times m \)矩阵， X为\(m \times k \)矩阵

参数：

- B (Tensor) – \(m \times k \)矩阵
- A (Tensor) – \(m \times m \)矩阵
- out (Tensor, optional) – 可选地输出矩阵\(X \)

例子:

```
>>> A = torch.Tensor([[6.80, -2.11,  5.66,  5.97,  8.23],
...                   [-6.05, -3.30,  5.36, -4.44,  1.08],
...                   [-0.45,  2.58, -2.70,  0.27,  9.04],
...                   [8.32,  2.71,  4.35,  -7.17,  2.14],
...                   [-9.67, -5.14, -7.26,  6.08, -6.87]]).t()
>>> B = torch.Tensor([[4.02,  6.19, -8.22, -7.57, -3.03],
...                   [-1.56,  4.00, -8.67,  1.75,  2.86],
...                   [9.81, -4.09, -4.57, -8.61,  8.99]]).t()
>>> X, LU = torch.gesv(B, A)
>>> torch.dist(B, torch.mm(A, X))
9.250057093890353e-06
```

torch.inverse

```
torch.inverse(input, out=None) → Tensor
```

对方阵输入 `input` 取逆。

注意 ：Irrespective of the original strides, the returned matrix will be transposed, i.e. with strides (1, m) instead of (m, 1)

参数 ：

- input (Tensor) – 输入2维张量
- out (Tensor, optional) – 输出张量

例子:

```
>>> x = torch.rand(10, 10)
>>> x

 0.7800  0.2267  0.7855  0.9479  0.5914  0.7119  0.4437  0.9131  0.1289  0.1982
 0.0045  0.0425  0.2229  0.4626  0.6210  0.0207  0.6338  0.7067  0.6381  0.8196
 0.8350  0.7810  0.8526  0.9364  0.7504  0.2737  0.0694  0.5899  0.8516  0.3883
 0.6280  0.6016  0.5357  0.2936  0.7827  0.2772  0.0744  0.2627  0.6326  0.9153
 0.7897  0.0226  0.3102  0.0198  0.9415  0.9896  0.3528  0.9397  0.2074  0.6980
 0.5235  0.6119  0.6522  0.3399  0.3205  0.5555  0.8454  0.3792  0.4927  0.6086
 0.1048  0.0328  0.5734  0.6318  0.9802  0.4458  0.0979  0.3320  0.3701  0.0909
 0.2616  0.3485  0.4370  0.5620  0.5291  0.8295  0.7693  0.1807  0.0650  0.8497
 0.1655  0.2192  0.6913  0.0093  0.0178  0.3064  0.6715  0.5101  0.2561  0.3396
 0.4370  0.4695  0.8333  0.1180  0.4266  0.4161  0.0699  0.4263  0.8865  0.2578
[torch.FloatTensor of size 10x10]

>>> x = torch.rand(10, 10)
>>> y = torch.inverse(x)
>>> z = torch.mm(x, y)
>>> z

 1.0000  0.0000  0.0000 -0.0000  0.0000  0.0000  0.0000  0.0000 -0.0000 -0.0000
 0.0000  1.0000 -0.0000  0.0000  0.0000  0.0000 -0.0000 -0.0000 -0.0000 -0.0000
 0.0000  0.0000  1.0000 -0.0000 -0.0000  0.0000  0.0000  0.0000 -0.0000 -0.0000
 0.0000  0.0000  0.0000  1.0000  0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000
 0.0000  0.0000 -0.0000 -0.0000  1.0000  0.0000  0.0000 -0.0000 -0.0000 -0.0000
 0.0000  0.0000  0.0000 -0.0000  0.0000  1.0000 -0.0000 -0.0000 -0.0000 -0.0000
 0.0000  0.0000  0.0000 -0.0000  0.0000  0.0000  1.0000  0.0000 -0.0000  0.0000
 0.0000  0.0000 -0.0000 -0.0000  0.0000  0.0000 -0.0000  1.0000 -0.0000  0.0000
-0.0000  0.0000 -0.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000  1.0000 -0.0000
-0.0000  0.0000 -0.0000 -0.0000 -0.0000  0.0000 -0.0000 -0.0000  0.0000  1.0000
[torch.FloatTensor of size 10x10]

>>> torch.max(torch.abs(z - torch.eye(10))) # Max nonzero
5.096662789583206e-07
```

torch.mm

```
torch.mm(mat1, mat2, out=None) → Tensor
```

对矩阵 `mat1` 和 `mat2` 进行相乘。 如果 `mat1` 是一个\(n \times m \) 张量， `mat2` 是一个 \(m \times p \) 张量，将会输出一个 \(n \times p \) 张量 `out` 。

参数 ：

- mat1 (Tensor) – 第一个相乘矩阵
- mat2 (Tensor) – 第二个相乘矩阵
- out (Tensor, optional) – 输出张量

例子:

```
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.mm(mat1, mat2)
 0.0519 -0.3304  1.2232
 4.3910 -5.1498  2.7571
[torch.FloatTensor of size 2x3]
```

torch.mv

```
torch.mv(mat, vec, out=None) → Tensor
```

对矩阵 `mat` 和向量 `vec` 进行相乘。 如果 `mat` 是一个\(n \times m \)张量，`vec` 是一个\(m \) 元 1维张量，将会输出一个\(n \) 元 1维张量。

参数 ：

- mat (Tensor) – 相乘矩阵
- vec (Tensor) – 相乘向量
- out (Tensor, optional) – 输出张量

例子:

```
>>> mat = torch.randn(2, 3)
>>> vec = torch.randn(3)
>>> torch.mv(mat, vec)
-2.0939
-2.2950
[torch.FloatTensor of size 2]
```

torch.orgqr

```
torch.orgqr()
```

torch.ormqr

```
torch.ormqr()
```

torch.potrf

```
torch.potrf()
```

torch.potri

```
torch.potri()
```

torch.potrs

```
torch.potrs()
```

torch.pstrf

```
torch.pstrf()
```

torch.qr

```
torch.qr(input, out=None) -> (Tensor, Tensor)
```

计算输入矩阵的QR分解：返回两个矩阵\(q \) ,\(r \)， 使得 \(x=q*r \) ，这里\(q \) 是一个半正交矩阵与 \(r \) 是一个上三角矩阵

本函数返回一个thin(reduced)QR分解。

注意 如果输入很大，可能可能会丢失精度。

注意 本函数依赖于你的LAPACK实现，虽然总能返回一个合法的分解，但不同平台可能得到不同的结果。

Irrespective of the original strides, the returned matrix q will be transposed, i.e. with strides (1, m) instead of (m, 1).

参数：

- input (Tensor) – 输入的2维张量
- out (tuple, optional) – 输出元组 `tuple` ，包含Q和R

例子:

```
>>> a = torch.Tensor([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> q, r = torch.qr(a)
>>> q

-0.8571  0.3943  0.3314
-0.4286 -0.9029 -0.0343
 0.2857 -0.1714  0.9429
[torch.FloatTensor of size 3x3]

>>> r

 -14.0000  -21.0000   14.0000
   0.0000 -175.0000   70.0000
   0.0000    0.0000  -35.0000
[torch.FloatTensor of size 3x3]

>>> torch.mm(q, r).round()

  12  -51    4
   6  167  -68
  -4   24  -41
[torch.FloatTensor of size 3x3]

>>> torch.mm(q.t(), q).round()

 1 -0  0
-0  1  0
 0  0  1
[torch.FloatTensor of size 3x3]
```

torch.svd

```
torch.svd(input, some=True, out=None) -> (Tensor, Tensor, Tensor)
```

\(U, S, V = torch.svd(A) \)。 返回对形如 \(n \times m \)的实矩阵 A 进行奇异值分解的结果，使得 \(A=USV'∗ \)。 \(U \) 形状为 \(n \times n \)，\(S \) 形状为 \(n \times m \) ，\(V \) 形状为 \(m \times m \)

`some` 代表了需要计算的奇异值数目。如果 `some=True` , it computes some and some=False computes all.

Irrespective of the original strides, the returned matrix U will be transposed, i.e. with strides (1, n) instead of (n, 1).

参数：

- input (Tensor) – 输入的2维张量
- some (bool, optional) – 布尔值，控制需计算的奇异值数目
- out (tuple, optional) – 结果 `tuple`

例子：

```
>>> a = torch.Tensor([[8.79,  6.11, -9.15,  9.57, -3.49,  9.84],
...                   [9.93,  6.91, -7.93,  1.64,  4.02,  0.15],
...                   [9.83,  5.04,  4.86,  8.83,  9.80, -8.99],
...                   [5.45, -0.27,  4.85,  0.74, 10.00, -6.02],
...                   [3.16,  7.98,  3.01,  5.80,  4.27, -5.31]]).t()
>>> a

  8.7900   9.9300   9.8300   5.4500   3.1600
  6.1100   6.9100   5.0400  -0.2700   7.9800
 -9.1500  -7.9300   4.8600   4.8500   3.0100
  9.5700   1.6400   8.8300   0.7400   5.8000
 -3.4900   4.0200   9.8000  10.0000   4.2700
  9.8400   0.1500  -8.9900  -6.0200  -5.3100
[torch.FloatTensor of size 6x5]

>>> u, s, v = torch.svd(a)
>>> u

-0.5911  0.2632  0.3554  0.3143  0.2299
-0.3976  0.2438 -0.2224 -0.7535 -0.3636
-0.0335 -0.6003 -0.4508  0.2334 -0.3055
-0.4297  0.2362 -0.6859  0.3319  0.1649
-0.4697 -0.3509  0.3874  0.1587 -0.5183
 0.2934  0.5763 -0.0209  0.3791 -0.6526
[torch.FloatTensor of size 6x5]

>>> s

 27.4687
 22.6432
  8.5584
  5.9857
  2.0149
[torch.FloatTensor of size 5]

>>> v

-0.2514  0.8148 -0.2606  0.3967 -0.2180
-0.3968  0.3587  0.7008 -0.4507  0.1402
-0.6922 -0.2489 -0.2208  0.2513  0.5891
-0.3662 -0.3686  0.3859  0.4342 -0.6265
-0.4076 -0.0980 -0.4932 -0.6227 -0.4396
[torch.FloatTensor of size 5x5]

>>> torch.dist(a, torch.mm(torch.mm(u, torch.diag(s)), v.t()))
8.934150226306685e-06
```

torch.symeig

```
torch.symeig(input, eigenvectors=False, upper=True, out=None) -> (Tensor, Tensor)
```

\(e, V = torch.symeig(input)\) 返回实对称矩阵 `input` 的特征值和特征向量。

\(input\) 和 \(V\) 为 \(m \times m\) 矩阵，\(e \) 是一个\(m\) 维向量。 此函数计算 `intput` 的所有特征值(和特征向量)，使得 \(input = V diag(e) V'\)
布尔值参数 `eigenvectors` 规定是否只计算特征向量。如果为 `False` ，则只计算特征值；若设为 `True` ，则两者都会计算。 因为输入矩阵 \(input\) 是对称的，所以默认只需要上三角矩阵。如果参数 `upper` 为 `False` ，下三角矩阵部分也被利用。

注意: 不管原来Irrespective of the original strides, the returned matrix V will be transposed, i.e. with strides (1, m) instead of (m, 1)

参数：

- input (Tensor) – 输入对称矩阵
- eigenvectors (boolean, optional) – 布尔值（可选），控制是否计算特征向量
- upper (boolean, optional) – 布尔值（可选），控制是否考虑上三角或下三角区域
- out (tuple, optional) – 输出元组(Tensor, Tensor)

例子：

```
>>> a = torch.Tensor([[ 1.96,  0.00,  0.00,  0.00,  0.00],
...                   [-6.49,  3.80,  0.00,  0.00,  0.00],
...                   [-0.47, -6.39,  4.17,  0.00,  0.00],
...                   [-7.20,  1.50, -1.51,  5.70,  0.00],
...                   [-0.65, -6.34,  2.67,  1.80, -7.10]]).t()

>>> e, v = torch.symeig(a, eigenvectors=True)
>>> e

-11.0656
 -6.2287
  0.8640
  8.8655
 16.0948
[torch.FloatTensor of size 5]

>>> v

-0.2981 -0.6075  0.4026 -0.3745  0.4896
-0.5078 -0.2880 -0.4066 -0.3572 -0.6053
-0.0816 -0.3843 -0.6600  0.5008  0.3991
-0.0036 -0.4467  0.4553  0.6204 -0.4564
-0.8041  0.4480  0.1725  0.3108  0.1622
[torch.FloatTensor of size 5x5]
```

torch.trtrs

```
torch.trtrs()
```

torch.Tensor

`torch.Tensor` 是一种包含单一数据类型元素的多维矩阵。

Torch定义了七种CPU tensor类型和八种GPU tensor类型：

Data tyoe	CPU tensor	GPU tensor
32-bit floating point	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	N/A	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

`torch.Tensor` 是默认的tensor类型（ `torch.FlaotTensor` ）的简称。

一个张量tensor可以从Python的 `list` 或序列构建：

```
>>> torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
1 2 3
4 5 6
[torch.FloatTensor of size 2x3]
```

一个空张量tensor可以通过规定其大小来构建：

```
>>> torch.IntTensor(2, 4).zero_()
0 0 0 0
0 0 0 0
[torch.IntTensor of size 2x4]
```

可以用python的索引和切片来获取和修改一个张量tensor中的内容：

```
>>> x = torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
>>> print(x[1][2])
6.0
>>> x[0][1] = 8
>>> print(x)
 1 8 3
 4 5 6
[torch.FloatTensor of size 2x3]
```

每一个张量tensor都有一个相应的`torch.Storage`用来保存其数据。类tensor提供了一个存储的多维的、横向视图，并且定义了在数值运算。

> ！注意： 会改变tensor的函数操作会用一个下划线后缀来标示。比如，`torch.FloatTensor.abs_()`会在原地计算绝对值，并返回改变后的tensor，而`tensor.FloatTensor.abs()`将会在一个新的tensor中计算结果。

```
class torch.Tensor
class torch.Tensor(*sizes)
class torch.Tensor(size)
class torch.Tensor(sequence)
class torch.Tensor(ndarray)
class torch.Tensor(tensor)
class torch.Tensor(storage)
```

根据可选择的大小和数据新建一个tensor。 如果没有提供参数，将会返回一个空的零维张量。如果提供了`numpy.ndarray`,`torch.Tensor`或`torch.Storage`，将会返回一个有同样参数的tensor.如果提供了python序列，将会从序列的副本创建一个tensor。

abs() → Tensor

请查看`torch.abs()`

abs_() → Tensor

`abs()`的in-place运算形式

acos() → Tensor

请查看`torch.acos()`

acos_() → Tensor

`acos()`的in-place运算形式

add(*value*)

请查看`torch.add()`

add(*_value*)

`add()`的in-place运算形式

addbmm(*beta=1, mat, alpha=1, batch1, batch2*) → Tensor

请查看`torch.addbmm()`

addbmm(*_beta=1, mat, alpha=1, batch1, batch2*) → Tensor

`addbmm()` 的in-place运算形式

addcdiv(*value=1, tensor1, tensor2*) → Tensor

请查看 `torch.addcdiv()`

addcdiv(*_value=1, tensor1, tensor2*) → Tensor

`addcdiv()` 的in-place运算形式

addcmul(*value=1, tensor1, tensor2*) → Tensor

请查看 `torch.addcmul()`

addcmul(*_value=1, tensor1, tensor2*) → Tensor

`addcmul()` 的in-place运算形式

addmm(*beta=1, mat, alpha=1, mat1, mat2*) → Tensor

请查看 `torch.addmm()`

addmm(*_beta=1, mat, alpha=1, mat1, mat2*) → Tensor

`addmm()` 的in-place运算形式

addmv(*beta=1, tensor, alpha=1, mat, vec*) → Tensor

请查看 `torch.addmv()`

addmv(*_beta=1, tensor, alpha=1, mat, vec*) → Tensor

`addmv()` 的in-place运算形式

addr(*beta=1, alpha=1, vec1, vec2*) → Tensor

请查看 `torch.addr()`

addr(*_beta=1, alpha=1, vec1, vec2*) → Tensor

`addr()` 的in-place运算形式

apply(*_callable*) → Tensor

将函数 `callable` 作用于tensor中每一个元素，并将每个元素用 `callable` 函数返回值替代。

> ！注意： 该函数只能在CPU tensor中使用，并且不应该用在有较高性能要求的代码块。
>
> ### asin() → Tensor

请查看 `torch.asin()`

asin_() → Tensor

`asin()` 的in-place运算形式

atan() → Tensor

请查看 `torch.atan()`

atan2() → Tensor

请查看 `torch.atan2()`

atan2_() → Tensor

`atan2()` 的in-place运算形式

atan_() → Tensor

`atan()` 的in-place运算形式

baddbmm(*beta=1, alpha=1, batch1, batch2*) → Tensor

请查看 `torch.baddbmm()`

baddbmm(*_beta=1, alpha=1, batch1, batch2*) → Tensor

`baddbmm()` 的in-place运算形式

bernoulli() → Tensor

请查看 `torch.bernoulli()`

bernoulli_() → Tensor

`bernoulli()` 的in-place运算形式

bmm(*batch2*) → Tensor

请查看 `torch.bmm()`

byte() → Tensor

将tensor改为byte类型

bmm(*median=0, sigma=1, *, generator=None*) → Tensor

将tensor中元素用柯西分布得到的数值填充：

> $$ P(x)={\frac1 \pi} {\frac \sigma {(x-median)^2 + \sigma^2}}
>
> $$
>
> #### ceil() → Tensor
>
> 请查看 `torch.ceil()`
>
> #### ceil_() → Tensor
>
> `ceil()` 的in-place运算形式
>
> #### char()
>
> 将tensor元素改为char类型
>
> #### chunk(*n_chunks, dim=0*) → Tensor
>
> 将tensor分割为tensor元组. 请查看 `torch.chunk()`
>
> #### clamp(*min, max*) → Tensor
>
> 请查看 `torch.clamp()`
>
> #### clamp(*_min, max*) → Tensor
>
> `clamp()` 的in-place运算形式
>
> #### clone() → Tensor
>
> 返回与原tensor有相同大小和数据类型的tensor
>
> #### contiguous() → Tensor
>
> 返回一个内存连续的有相同数据的tensor，如果原tensor内存连续则返回原tensor
>
> #### copy(*_src, async=False*) → Tensor
>
> 将 `src` 中的元素复制到tensor中并返回这个tensor。 两个tensor应该有相同数目的元素，可以是不同的数据类型或存储在不同的设备上。 参数：
>
> - **src** (*Tensor*)-复制的源tensor
> - **async** (*bool*)-如果为True并且复制是在CPU和GPU之间进行的，则复制后的拷贝可能会与源信息异步，对于其他类型的复制操作则该参数不会发生作用。

cos() → Tensor

请查看 `torch.cos()`

cos_() → Tensor

`cos()` 的in-place运算形式

cosh() → Tensor

请查看 `torch.cosh()`

cosh_() → Tensor

`cosh()` 的in-place运算形式

cpu() → Tensor

如果在CPU上没有该tensor，则会返回一个CPU的副本

cross(*other, dim=-1*) → Tensor

请查看 `torch.cross()`

cuda(device=None, async=False)

返回此对象在CPU内存中的一个副本 如果对象已近存在与CUDA存储中并且在正确的设备上，则不会进行复制并返回原始对象。

参数：

- **device**(*int*)-目的GPU的id，默认为当前的设备。
- **async**(*bool*)-如果为True并且资源在固定内存中，则复制的副本将会与原始数据异步。否则，该参数没有意义。

cumprod(*dim*) → Tensor

请查看 `torch.cumprod()`

cumsum(*dim*) → Tensor

请查看 `torch.cumsum()`

data_ptr() → int

返回tensor第一个元素的地址

diag(*diagonal=0*) → Tensor

请查看 `torch.diag()`

dim() → int

返回tensor的维数

dist(*other*, *p*=2) → Tensor

请查看 `torch.dist()`

div(*value*)

请查看 `torch.div()`

div(*_value*)

`div()` 的in-place运算形式

dot(*tensor2*) → float

请查看 `torch.dot()`

double()

将该tensor投射为double类型

eig(*eigenvectors=False*) -> (*Tensor, Tensor*)

请查看 `torch.eig()`

element_size() → int

返回单个元素的字节大小。 例：

```
>>> torch.FloatTensor().element_size()
4
>>> torch.ByteTensor().element_size()
1
```

eq(*other*) → Tensor

请查看 `torch.eq()`

eq(*_other*) → Tensor

`eq()` 的in-place运算形式

equal(*other*) → bool

请查看 `torch.equal()`

exp() → Tensor

请查看 `torch.exp()`

exp_() → Tensor

`exp()` 的in-place运算形式

expand(*sizes)

返回tensor的一个新视图，单个维度扩大为更大的尺寸。 tensor也可以扩大为更高维，新增加的维度将附在前面。 扩大tensor不需要分配新内存，只是仅仅新建一个tensor的视图，其中通过将 `stride` 设为0，一维将会扩展位更高维。任何一个一维的在不分配新内存情况下可扩展为任意的数值。

参数：

- **sizes**(*torch.Size or int...*)-需要扩展的大小

例：

```
>>> x = torch.Tensor([[1], [2], [3]])
>>> x.size()
torch.Size([3, 1])
>>> x.expand(3, 4)
 1 1
 1 1
 2 2 2 2
 3 3 3 3
 [torch.FloatTensor of size 3x4]
```

expand*as(_tensor*)

将tensor扩展为参数tensor的大小。 该操作等效与：

```
self.expand(tensor.size())
```

exponential*(_lambd=1, *, generator=None*) to Tensor

将该tensor用指数分布得到的元素填充：

$$ P(x)= \lambda e^{- \lambda x}

$$

fill*(_value*) → Tensor

将该tensor用指定的数值填充

float()

将tensor投射为float类型

floor() → Tensor

请查看 `torch.floor()`

floor_() → Tensor

`floor()` 的in-place运算形式

fmod(*divisor*) → Tensor

请查看 `torch.fmod()`

fmod(*_divisor*) → Tensor

`fmod()` 的in-place运算形式

frac() → Tensor

请查看 `torch.frac()`

frac_() → Tensor

`frac()` 的in-place运算形式

gather(*dim, index*) → Tensor

请查看 `torch.gather()`

ge(*other*) → Tensor

请查看 `torch.ge()`

ge(*_other*) → Tensor

`ge()` 的in-place运算形式

gels(*A*) → Tensor

请查看 `torch.gels()`

geometric(*_p, *, generator=None*) → Tensor

将该tensor用几何分布得到的元素填充：

$$ P(X=k)= (1-p)^{k-1}p $$

geqrf() -> (*Tensor, Tensor*)

请查看 `torch.geqrf()`

ger(*vec2*) → Tensor

请查看 `torch.ger()`

gesv(*A*) → Tensor, Tensor

请查看 `torch.gesv()`

gt(*other*) → Tensor

请查看 `torch.gt()`

gt(*_other*) → Tensor

`gt()` 的in-place运算形式

half()

将tensor投射为半精度浮点类型

histc(*bins=100, min=0, max=0*) → Tensor

请查看 `torch.histc()`

index(*m*) → Tensor

用一个二进制的掩码或沿着一个给定的维度从tensor中选取元素。`tensor.index(m)` 与 `tensor[m]` 完全相同。

参数：

- **m**(*int or Byte Tensor or slice*)-用来选取元素的维度或掩码

  ### index*add*(*dim, index, tensor*) → Tensor

  按参数index中的索引数确定的顺序，将参数tensor中的元素加到原来的tensor中。参数tensor的尺寸必须严格地与原tensor匹配，否则会发生错误。

参数：

- **dim**(*int*)-索引index所指向的维度
- **index**(*LongTensor*)-需要从tensor中选取的指数
- **tensor**(*Tensor*)-含有相加元素的tensor

例：

```
>>> x = torch.Tensor([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
>>> t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2, 1])
>>> x.index_add_(0, index, t)
>>> x
  2 3 4
  8 9 10
  5 6 7
[torch.FloatTensor of size 3x3]
```

index*copy*(*dim, index, tensor*) → Tensor

按参数index中的索引数确定的顺序，将参数tensor中的元素复制到原来的tensor中。参数tensor的尺寸必须严格地与原tensor匹配，否则会发生错误。

参数：

- **dim** (*int*)-索引index所指向的维度
- **index** (*LongTensor*)-需要从tensor中选取的指数
- **tensor** (*Tensor*)-含有被复制元素的tensor

例：

```
>>> x = torch.Tensor(3， 3)
>>> t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2, 1])
>>> x.index_copy_(0, index, t)
>>> x
  1 2 3
  7 8 9
  4 5 6
[torch.FloatTensor of size 3x3]
```

index*fill*(*dim, index, val*) → Tensor

按参数index中的索引数确定的顺序，将原tensor用参数 `val` 值填充。

参数：

- **dim** (*int*)-索引index所指向的维度
- **index** (*LongTensor*)-索引
- **val** (*Tensor*)-填充的值

例：

```
>>> x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2])
>>> x.index_fill_(0, index, -1)
>>> x
  -1 2 -1
  -1 5 -1
  -1 8 -1
[torch.FloatTensor of size 3x3]
```

index*select(_dim, index*) → Tensor

请查看 `torch.index_select()`

int()

将该tensor投射为int类型

inverse() → Tensor

请查看 `torch.inverse()`

is_contiguous() → bool

如果该tensor在内存中是连续的则返回True。

is_cuda

is_pinned()

如果该tensor在固定内内存中则返回True

is*set_to(_tensor*) → bool

如果此对象引用与Torch C API相同的 `THTensor` 对象作为给定的张量，则返回True。

is_signed()

kthvalue(*k, dim=None*) -> (*Tensor, LongTensor*)

请查看 `torch.kthvalue()`

le(*other*) → Tensor

请查看 `torch.le()`

le*(_other*) → Tensor

`le()` 的in-place运算形式

lerp(*start, end, weight*)

请查看 `torch.lerp()`

lerp_(*start, end, weight*) → Tensor

`lerp()` 的in-place运算形式

log() → Tensor

请查看 `torch.log()`

loglp() → Tensor

请查看 `torch.loglp()`

loglp_() → Tensor

`loglp()` 的in-place运算形式

log_()→ Tensor

`log()` 的in-place运算形式

log*normal*(*mwan=1, std=2,* , gegnerator=None*)

将该tensor用均值为μ,标准差为σ的对数正态分布得到的元素填充。要注意 `mean` 和 `stdv` 是基本正态分布的均值和标准差，不是返回的分布：

$$ P(X)= \frac {1} {x \sigma \sqrt {2 \pi}}e^{- \frac {(lnx- \mu)^2} {2 \sigma^2}}

$$

long()

将tensor投射为long类型

lt(*other*) → Tensor

请查看 `torch.lt()`

lt_(*other*) → Tensor

`lt()` 的in-place运算形式

map_(*tensor, callable*)

将 `callable` 作用于本tensor和参数tensor中的每一个元素，并将结果存放在本tensor中。 `callable` 应该有下列标志：

```
def callable(a, b) -> number
```

masked*copy*(*mask, source*)

将 `mask` 中值为1元素对应的 `source` 中位置的元素复制到本tensor中。 `mask` 应该有和本tensor相同数目的元素。 `source` 中元素的个数最少为 `mask` 中值为1的元素的个数。

参数：

- **mask** (*ByteTensor*)-二进制掩码
- **source** (*Tensor*)-复制的源tensor

> 注意：`mask`作用于`self`自身的tensor，而不是参数中的`source`。
>
> #### masked*fill*(*mask, value*)
>
> 在`mask`值为1的位置处用`value`填充。`mask`的元素个数需和本tensor相同，但尺寸可以不同。

参数：

- **mask** (*ByteTensor*)-二进制掩码
- **value** (*Tensor*)-用来填充的值

masked_select(*mask*) → Tensor

请查看`torch.masked_select()`

max(*dim=None*) -> *float or(Tensor, Tensor)*

请查看`torch.max()`

mean(*dim=None*) -> *float or(Tensor, Tensor)*

请查看`torch.mean()`

median(*dim=-1, value=None, indices=None*) -> *(Tensor, LongTensor)*

请查看`torch.median()`

min(*dim=None*) -> *float or(Tensor, Tensor)*

请查看`torch.min()`

mm(*mat2*) → Tensor

请查看`torch.mm()`

mode(*dim=-1, value=None, indices=None*) -> *(Tensor, LongTensor)*

请查看`torch.mode()`

mul(*value*) → Tensor

请查看`torch.mul()`

mul_(*value*)

`mul()` 的in-place运算形式

multinomial(*num_samples, replacement=False,* , generator=None*) → Tensor

请查看 `torch.multinomial()`

mv(*vec*) → Tensor

请查看 `torch.mv()`

narrow(*dimension, start, length*) → Te

返回一个本tensor经过缩小后的tensor。维度 `dim` 缩小范围是 `start` 到 `start+length` 。原tensor与返回的tensor共享相同的底层内存。

参数：

- **dimension** (*int*)-需要缩小的维度
- **start** (*int*)-起始维度
- **length** (*int*)-

例:

```
>>> x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x.narrow(0, 0, 2)
 1  2  3
 4  5  6
[torch.FloatTensor of size 2x3]
>>> x.narrow(1, 1, 2)
 2  3
 5  6
 8  9
[torch.FloatTensor of size 3x2]
```

ndimension() → int

`dim()` 的另一种表示。

ne(*other*) → Tensor

请查看 `torch.ne()`

ne_(*other*) → Tensor

`ne()` 的in-place运算形式

neg() → Tensor

请查看 `torch.neg()`

neg_() → Tensor

`neg()` 的in-place运算形式

nelement() → int

`numel()` 的另一种表示

new(*args, *kwargs*)

构建一个有相同数据类型的tensor

nonezero() → LongTensor

请查看`torch.nonezero()

norm(*p=2*) → float

请查看`torch.norm()

normal_(*mean=0, std=1,* , gengerator=None*)

将tensor用均值为 `mean` 和标准差为 `std` 的正态分布填充。

numel() → int

请查看 `numel()`

numpy() → ndarray

将该tensor以NumPy的形式返回 `ndarray` ，两者共享相同的底层内存。原tensor改变后会相应的在 `ndarray` 有反映，反之也一样。

orgqr(*input2*) → Tensor

请查看 `torch.orgqr()`

ormqr(*input2, input3, left=True, transpose=False*) → Tensor

请查看 `torch.ormqr()`

permute(*dims*)

将tensor的维度换位。

参数：

- ***dims** (*int..*)-换位顺序

例：

```
>>> x = torch.randn(2, 3, 5)
>>> x.size()
torch.Size([2, 3, 5])
>>> x.permute(2, 0, 1).size()
torch.Size([5, 2, 3])
```

pin_memory()

如果原来没有在固定内存中，则将tensor复制到固定内存中。

potrf(*upper=True*) → Tensor

请查看 `torch.potrf()`

potri(*upper=True*) → Tensor

请查看 `torch.potri()`

potrs(*input2, upper=True*) → Tensor

请查看 `torch.potrs()`

pow(*exponent*)

请查看 `torch.pow()`

pow_()

`pow()` 的in-place运算形式

prod()) → float

请查看 `torch.prod()`

pstrf(*upper=True, tol=-1*) -> (*Tensor, IntTensor*)

请查看 `torch.pstrf()`

qr()-> (*Tensor, IntTensor*)

请查看 `torch.qr()`

random(*_from=0, to=None, *, generator=None*)

将tensor用从在[from, to-1]上的正态分布或离散正态分布取样值进行填充。如果没有明确说明，则填充值仅由本tensor的数据类型限定。

reciprocal() → Tensor

请查看 `torch.reciprocal()`

reciprocal_() → Tensor

`reciprocal()` 的in-place运算形式

remainder(*divisor*) → Tensor

请查看 `torch.remainder()`

remainder_(*divisor*) → Tensor

`remainder()` 的in-place运算形式

renorm(*p, dim, maxnorm*) → Tensor

请查看 `torch.renorm()`

renorm_(*p, dim, maxnorm*) → Tensor

`renorm()` 的in-place运算形式

repeat(**sizes*)

沿着指定的维度重复tensor。 不同于 `expand()` ，本函数复制的是tensor中的数据。

参数：

- ***sizes** (*torch.Size ot int...*)-沿着每一维重复的次数

例：

```
>>> x = torch.Tensor([1, 2, 3])
>>> x.repeat(4, 2)
 1  2  3  1  2  3
 1  2  3  1  2  3
 1  2  3  1  2  3
 1  2  3  1  2  3
[torch.FloatTensor of size 4x6]
>>> x.repeat(4, 2, 1).size()
torch.Size([4, 2, 3])
```

resize(*sizes_)

将tensor的大小调整为指定的大小。如果元素个数比当前的内存大小大，就将底层存储大小调整为与新元素数目一致的大小。如果元素个数比当前内存小，则底层存储不会被改变。原来tensor中被保存下来的元素将保持不变，但新内存将不会被初始化。

参数：

- ***sizes** (*torch.Size or int...*)-需要调整的大小

例：

```
>>> x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
>>> x.resize_(2, 2)
>>> x
 1  2
 3  4
[torch.FloatTensor of size 2x2]
```

resize*as*(*tensor*)

将本tensor的大小调整为与参数中的tensor相同的大小。等效于：

```
self.resize_(tensor.size())
```

round() → Tensor

请查看 `torch.round()`

round_() → Tensor

`round()` 的in-place运算形式

rsqrt() → Tensor

请查看 `torch.rsqrt()`

rsqrt_() → Tensor

`rsqrt()` 的in-place运算形式

scatter_(*input, dim, index, src*) → Tensor

将 `src` 中的所有值按照 `index` 确定的索引写入本tensor中。其中索引是根据给定的dimension，dim按照 `gather()` 描述的规则来确定。

注意，index的值必须是在*0*到*(self.size(dim)-1)*之间，

参数：

- **input** (*Tensor*)-源tensor
- **dim** (*int*)-索引的轴向
- **index** (*LongTensor*)-散射元素的索引指数
- **src** (*Tensor or float*)-散射的源元素

例：

```
>>> x = torch.rand(2, 5)
>>> x

 0.4319  0.6500  0.4080  0.8760  0.2355
 0.2609  0.4711  0.8486  0.8573  0.1029
[torch.FloatTensor of size 2x5]

>>> torch.zeros(3, 5).scatter_(0, torch.LongTensor([[0, 1, 2, 0, 0], [2, 0, 0, 1, 2]])
, x)

 0.4319  0.4711  0.8486  0.8760  0.2355
 0.0000  0.6500  0.0000  0.8573  0.0000
 0.2609  0.0000  0.4080  0.0000  0.1029
[torch.FloatTensor of size 3x5]

>>> z = torch.zeros(2, 4).scatter_(1, torch.LongTensor([[2], [3]]), 1.23)
>>> z

 0.0000  0.0000  1.2300  0.0000
 0.0000  0.0000  0.0000  1.2300
[torch.FloatTensor of size 2x4]
```

select(*dim, index*) → Tensor or number

按照index中选定的维度将tensor切片。如果tensor是一维的，则返回一个数字。否则，返回给定维度已经被移除的tensor。

参数：

- **dim** (*int*)-切片的维度
- **index** (*int*)-用来选取的索引

> !注意： `select()` 等效于切片。例如， `tensor.select(0, index)` 等效于 `tensor[index]` ， `tensor.select(2, index)` 等效于 `tensor[:, :, index]` .
>
> ### set(*source=None, storage_offset=0, size=None, stride=None*)
>
> 设置底层内存，大小和步长。如果 `tensor` 是一个tensor，则将会与本tensor共享底层内存并且有相同的大小和步长。改变一个tensor中的元素将会反映在另一个tensor。 如果 `source` 是一个 `Storage` ，则将设置底层内存，偏移量，大小和步长。

参数：

- **source** (*Tensor or Storage*)-用到的tensor或内存
- **storage_offset** (*int*)-内存的偏移量
- **size** (*torch.Size*)-需要的大小，默认为源tensor的大小。
- **stride**(*tuple*)-需要的步长，默认为C连续的步长。

share*memory*()

将底层内存移到共享内存中。 如果底层内存已经在共享内存中是将不进行任何操作。在共享内存中的tensor不能调整大小。

short()

将tensor投射为short类型。

sigmoid() → Tensor

请查看 `torch.sigmoid()`

sigmoid_() → Tensor

`sidmoid()` 的in-place运算形式

sign() → Tensor

请查看 `torch.sign()`

sign_() → Tensor

`sign()` 的in-place运算形式

sin() → Tensor

请查看 `torch.sin()`

sin_() → Tensor

`sin()` 的in-place运算形式

sinh() → Tensor

请查看 `torch.sinh()`

sinh_() → Tensor

`sinh()` 的in-place运算形式

size() → torch.Size

返回tensor的大小。返回的值是 `tuple` 的子类。

例：

```
>>> torch.Tensor(3, 4, 5).size()
torch.Size([3, 4, 5])
```

sort(*dim=None, descending=False*) -> (*Tensor, LongTensor*)

请查看 `torhc.sort()`

split(*split_size, dim=0*)

将tensor分割成tensor数组。 请查看 `torhc.split()`

sqrt() → Tensor

请查看 `torch.sqrt()`

sqrt_() → Tensor

`sqrt()` 的in-place运算形式

squeeze(*dim=None*) → Tensor

请查看 `torch.squeeze()`

squeeze(*_dim=None*) → Tensor

`squeeze()` 的in-place运算形式

std() → float

请查看 `torch.std()`

storage() → torch.Storage

返回底层内存。

storage_offset() → int

以储存元素的个数的形式返回tensor在地城内存中的偏移量。 例：

```
>>> x = torch.Tensor([1, 2, 3, 4, 5])
>>> x.storage_offset()
0
>>> x[3:].storage_offset()
3
```

classmethod() storage_type()

stride() → Tensor

返回tesnor的步长。

sub(*value, other*) → Tensor

从tensor中抽取一个标量或tensor。如果 `value` 和 `other` 都是给定的，则在使用之前 `other` 的每一个元素都会被 `value` 缩放。

sub(*_x*) → Tensor

`sub()` 的in-place运算形式

sum(*dim*=*None*) → Tensor

请查看 `torch.sum()`

svd(*some*=*True*) -> (Tensor, Tensor, Tensor)

请查看 `torch.svd()`

symeig(_eigenvectors=False, upper=True) -> (Tensor, Tensor)

请查看 `torch.symeig()`

t() → Tensor

请查看 `torch.t()`

t() → Tensor

`t()` 的in-place运算形式

tan() → Tensor

请查看 `torch.tan()`

tan_() → Tensor

`tan()` 的in-place运算形式

tanh() → Tensor

请查看 `torch.tanh()`

tanh_() → Tensor

`tanh()` 的in-place运算形式

tolist()

返回一个tensor的嵌套列表表示。

topk(*k, dim*=*None, largest*=*True, sorted*=*True*) -> (Tensor, LongTensor)

请查看 `torch.topk()`

trace() → float

请查看 `torch.trace()`

transpose(*dim0, dim1*) → Tensor

请查看 `torch.transpose()`

transpose(*dim0, dim1*) → Tensor

`transpose()` 的in-place运算形式

tril(*k=0*) → Tensor

请查看 `torch.tril()`

tril(*_k=0*) → Tensor

`tril()` 的in-place运算形式

triu(*k=0*) → Tensor

请查看 `torch.triu()`

triu(*k=0*) → Tensor

`triu()` 的in-place运算形式

trtrs(*A, upper=True, transpose=False, unitriangular=False*) -> (Tensor, Tensor)

请查看 `torch.trtrs()`

trunc() → Tensor

请查看 `torch.trunc()`

trunc() → Tensor

`trunc()` 的in-place运算形式

type(*new_type=None, async=False*)

将对象投为指定的类型。 如果已经是正确的类型，则不会进行复制并返回原对象。

参数：

- **new_type** (*type or string*)-需要的类型
- **async** (*bool*)-如果为True，并且源地址在固定内存中，目的地址在GPU或者相反，则会相对于源主异步执行复制。否则，该参数不发挥作用。

type*as(_tesnor*)

将tensor投射为参数给定tensor类型并返回。 如果tensor已经是正确的类型则不会执行操作。等效于：

```
self.type(tensor.type())
```

参数：

- **tensor** (Tensor):有所需要类型的tensor

unfold(*dim, size, step*) → Tensor

返回一个tensor，其中含有在`dim`维tianchong度上所有大小为`size`的分片。两个分片之间的步长为`step`。 如果*sizedim*是dim维度的原始大小，则在返回tensor中的维度dim大小是*(sizedim-size)/step+1* 维度大小的附加维度将附加在返回的tensor中。

参数：

- **dim** (*int*)-需要展开的维度
- **size** (*int*)-每一个分片需要展开的大小
- **step** (*int*)-相邻分片之间的步长

例：

```
>>> x = torch.arange(1, 8)
>>> x

 1
 2
 3
 4
 5
 6
 7
[torch.FloatTensor of size 7]

>>> x.unfold(0, 2, 1)

 1  2
 2  3
 3  4
 4  5
 5  6
 6  7
[torch.FloatTensor of size 6x2]

>>> x.unfold(0, 2, 2)

 1  2
 3  4
 5  6
[torch.FloatTensor of size 3x2]
```

uniform(*_from=0, to=1*) → Tensor

将tensor用从均匀分布中抽样得到的值填充。

unsqueeze(*dim*)

请查看 `torch.unsqueeze()`

unsqueeze(*_dim*) → Tensor

`unsqueeze()` 的in-place运算形式

var()

请查看 `torch.var()`

view(**args*) → Tensor

返回一个有相同数据但大小不同的tensor。 返回的tensor必须有与原tensor相同的数据和相同数目的元素，但可以有不同的大小。一个tensor必须是连续的 `contiguous()` 才能被查看。

例：

```
>>> x = torch.randn(4, 4)
>>> x.size()
torch.Size([4, 4])
>>> y = x.view(16)
>>> y.size()
torch.Size([16])
>>> z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
>>> z.size()
torch.Size([2, 8])
```

view*as(_tensor*)

返回被视作与给定的tensor相同大小的原tensor。 等效于：

```
self.view(tensor.size())
```

zero_()

用0填充该tensor。

torch.Storage

一个 `torch.Storage` 是一个单一数据类型的连续一维数组。

每个 `torch.Tensor` 都有一个对应的、相同数据类型的存储。

```
class torch.FloatStorage
```

byte()

将此存储转为byte类型

char()

将此存储转为char类型

clone()

返回此存储的一个副本

copy_()

cpu()

如果当前此存储不在CPU上，则返回一个它的CPU副本

cuda(*device=None, async=False*)

返回此对象在CUDA内存中的一个副本。
如果此对象已在CUDA内存中且在正确的设备上，那么不会执行复制操作，直接返回原对象。

参数：

- **device** (*int*) - 目标GPU的id。默认值是当前设备。
- **async** (*bool*) -如果值为True，且源在锁定内存中，则副本相对于宿主是异步的。否则此参数不起效果。

data_ptr()

double()

将此存储转为double类型

element_size()

fill_()

float()

将此存储转为float类型

from_buffer()

half()

将此存储转为half类型

int()

将此存储转为int类型

is_cuda = *False*

is_pinned()

is_shared()

is_sparse = *False*

long()

将此存储转为long类型

new()

pin_memory()

如果此存储当前未被锁定，则将它复制到锁定内存中。

resize_()

share*memory*()

将此存储移动到共享内存中。
对于已经在共享内存中的存储或者CUDA存储，这是一条空指令，它们不需要移动就能在进程间共享。共享内存中的存储不能改变大小。
返回：self

short()

将此存储转为short类型

size()

tolist()

返回一个包含此存储中元素的列表

type(*new_type=None, async=False*)

将此对象转为指定类型。
如果已经是正确类型，不会执行复制操作，直接返回原对象。

参数：

- **new_type** (*type* or *string*) -需要转成的类型
- **async** (*bool*) -如果值为True，且源在锁定内存中而目标在GPU中——或正好相反，则复制操作相对于宿主异步执行。否则此参数不起效果。

torch.nn

Parameters

class torch.nn.Parameter()

`Variable` 的一种，常被用于模块参数(`module parameter`)。

`Parameters` 是 `Variable` 的子类。`Paramenters` 和 `Modules` 一起使用的时候会有一些特殊的属性，即：当 `Paramenters` 赋值给 `Module` 的属性的时候，他会自动的被加到 `Module` 的 参数列表中(即：会出现在 `parameters()` 迭代器中)。将 `Varibale` 赋值给 `Module` 属性则不会有这样的影响。 这样做的原因是：我们有时候会需要缓存一些临时的状态(`state`), 比如：模型中 `RNN` 的最后一个隐状态。如果没有 `Parameter` 这个类的话，那么这些临时变量也会注册成为模型变量。

`Variable` 与 `Parameter` 的另一个不同之处在于，`Parameter` 不能被 `volatile` (即：无法设置 `volatile=True`)而且默认 `requires_grad=True` 。 `Variable` 默认 `requires_grad=False` 。

参数说明:

- data (Tensor) – parameter tensor.
- requires_grad (bool, optional) – 默认为 `True` ，在 `BP` 的过程中会对其求微分。

Containers（容器）：

class torch.nn.Module

所有网络的基类。

你的模型也应该继承这个类。

`Modules` 也可以包含其它 `Modules` ,允许使用树结构嵌入他们。你可以将子模块赋值给模型属性。

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)# submodule: Conv2d
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
       x = F.relu(self.conv1(x))
       return F.relu(self.conv2(x))
```

通过上面方式赋值的 `submodule` 会被注册。当调用 `.cuda()` 的时候，`submodule` 的参数也会转换为 `cuda Tensor` 。

add_module(name, module)

将一个 `child module` 添加到当前 `modle` 。 被添加的 `module` 可以通过 `name` 属性来获取。 例：

```
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.add_module("conv", nn.Conv2d(10, 20, 4))
        #self.conv = nn.Conv2d(10, 20, 4) 和上面这个增加module的方式等价
model = Model()
print(model.conv)
```

输出：

```
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
```

children()

Returns an iterator over immediate children modules. 返回当前模型 子模块的迭代器。

```
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.add_module("conv", nn.Conv2d(10, 20, 4))
        self.add_module("conv1", nn.Conv2d(20 ,10, 4))
model = Model()

for sub_module in model.children():
    print(sub_module)
```

```
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
Conv2d(20, 10, kernel_size=(4, 4), stride=(1, 1))
```

cpu(device_id=None)

将所有的模型参数(`parameters`)和`buffers`复制到`CPU`

`NOTE`：官方文档用的move，但我觉着`copy`更合理。

cuda(device_id=None)

将所有的模型参数(`parameters`)和`buffers`赋值`GPU`

参数说明:

- device_id (int, optional) – 如果指定的话，所有的模型参数都会复制到指定的设备上。

double()

将`parameters`和`buffers`的数据类型转换成`double`。

eval()

将模型设置成`evaluation`模式

仅仅当模型中有`Dropout`和`BatchNorm`是才会有影响。

float()

将`parameters`和`buffers`的数据类型转换成`float`。

forward(* input)

定义了每次执行的 计算步骤。 在所有的子类中都需要重写这个函数。

half()

将`parameters`和`buffers`的数据类型转换成`half`。

load_state_dict(state_dict)

将`state_dict`中的`parameters`和`buffers`复制到此`module`和它的后代中。`state_dict`中的`key`必须和 `model.state_dict()`返回的`key`一致。 `NOTE`：用来加载模型参数。

参数说明:

- state_dict (dict) – 保存`parameters`和`persistent buffers`的字典。

modules()

返回一个包含 当前模型 所有模块的迭代器。

```
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.add_module("conv", nn.Conv2d(10, 20, 4))
        self.add_module("conv1", nn.Conv2d(20 ,10, 4))
model = Model()

for module in model.modules():
    print(module)
```

```
Model (
  (conv): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
  (conv1): Conv2d(20, 10, kernel_size=(4, 4), stride=(1, 1))
)
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
Conv2d(20, 10, kernel_size=(4, 4), stride=(1, 1))
```

可以看出，`modules()` 返回的 `iterator` 不止包含 子模块。这是和 `children()` 的不同。

`NOTE：` 重复的模块只被返回一次(`children()也是`)。 在下面的例子中, `submodule` 只会被返回一次：

```
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        submodule = nn.Conv2d(10, 20, 4)
        self.add_module("conv", submodule)
        self.add_module("conv1", submodule)
model = Model()

for module in model.modules():
    print(module)
```

```
Model (
  (conv): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
  (conv1): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
)
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
```

named_children()

返回 包含 模型当前子模块 的迭代器，`yield` 模块名字和模块本身。

例子：

```
for name, module in model.named_children():
    if name in ['conv4', 'conv5']:
        print(module)
```

named_modules(memo=None, prefix='')[source]

返回包含网络中所有模块的迭代器, `yielding` 模块名和模块本身。

`注意：`

重复的模块只被返回一次(`children()也是`)。 在下面的例子中, `submodule` 只会被返回一次。

parameters(memo=None)

返回一个 包含模型所有参数 的迭代器。

一般用来当作 `optimizer` 的参数。

例子：

```
for param in model.parameters():
    print(type(param.data), param.size())

<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(hook)

在 `module` 上注册一个 `bachward hook` 。

每次计算 `module` 的 `inputs` 的梯度的时候，这个 `hook` 会被调用。 `hook` 应该拥有下面的 `signature` 。

`hook(module, grad_input, grad_output) -> Variable or None`

如果 `module` 有多个输入输出的话，那么 `grad_input` `grad_output` 将会是个 `tuple` 。 `hook` 不应该修改它的 `arguments` ，但是它可以选择性的返回关于输入的梯度，这个返回的梯度在后续的计算中会替代 `grad_input` 。

这个函数返回一个 句柄(`handle`)。它有一个方法 `handle.remove()` ，可以用这个方法将 `hook` 从 `module` 移除。

register_buffer(name, tensor)

给 `module` 添加一个 `persistent buffer` 。

`persistent buffer` 通常被用在这么一种情况：我们需要保存一个状态，但是这个状态不能看作成为模型参数。 例如：, `BatchNorm’s` running_mean 不是一个 `parameter` , 但是它也是需要保存的状态之一。

`Buffers` 可以通过注册时候的 `name` 获取。

`NOTE` **:**我们可以用 **buffer** 保存 **`moving average`**

例子：

```
self.register_buffer('running_mean', torch.zeros(num_features))

self.running_mean
```

register_forward_hook(hook)

在 `module` 上注册一个 `forward hook` 。 每次调用 `forward()` 计算输出的时候，这个 `hook` 就会被调用 。它应该拥有以下签名：

`hook(module, input, output) -> None`

`hook` 不应该修改 `input` 和 `output` 的值。 这个函数返回一个 句柄(`handle`)。它有一个方法 `handle.remove()` ，可以用这个方法将 `hook` 从 `module` 移除。

register_parameter(name, param)

向 `module` 添加 `parameter`

`parameter` 可以通过注册时候的 `name` 获取。

state_dict(destination=None, prefix='')[source]

返回一个字典，保存着 `module` 的所有状态（ `state` ）。

`parameters` 和 `persistent buffers` 都会包含在字典中，字典的 `key` 就是 `parameter` 和 `buffer` 的 `names` 。

例子：

```
import torch
from torch.autograd import Variable
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv2 = nn.Linear(1, 2)
        self.vari = Variable(torch.rand([1]))
        self.par = nn.Parameter(torch.rand([1]))
        self.register_buffer("buffer", torch.randn([2,3]))

model = Model()
print(model.state_dict().keys())
```

```
odict_keys(['par', 'buffer', 'conv2.weight', 'conv2.bias'])
```

train(mode=True)

将 `module` 设置为 `training mode` 。

仅仅当模型中有 `Dropout` 和 `BatchNorm` 是才会有影响。

zero_grad()

将 `module` 中的所有模型参数的梯度设置为0.

class torch.nn.Sequential(* args)

一个时序容器。`Modules` 会以他们传入的顺序被添加到容器中。当然，也可以传入一个 `OrderedDict` 。

为了更容易的理解如何使用 `Sequential` , 下面给出了一个例子:

```
# Example of using Sequential

model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
# Example of using Sequential with OrderedDict
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

class torch.nn.ModuleList(modules=None)[source]

将 `submodules` 保存在一个 `list` 中。

`ModuleList` 可以像一般的 `Python list` 一样被 `索引` 。而且 `ModuleList` 中包含的 `modules` 已经被正确的注册，对所有的 `module method` 可见。

参数说明:

- modules (list, optional) – 将要被添加到 `MuduleList` 中的 `modules` 列表

例子:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed         using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

append(module)[source]

等价于 list 的 `append()`

参数说明:

- module (nn.Module) – 要 append 的 `module`

extend(modules)[source]

等价于 `list` 的 `extend()` 方法

参数说明:

- modules (list) – list of modules to append

class torch.nn.ParameterList(parameters=None)

将 `submodules` 保存在一个 `list` 中。

`ParameterList` 可以像一般的 `Python list` 一样被 `索引` 。而且 `ParameterList` 中包含的 `parameters` 已经被正确的注册，对所有的 `module method` 可见。

参数说明:

- modules (list, optional) – a list of nn.Parameter

例子:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.params = nn.ParameterList([nn.Parameter(torch.randn(10, 10)) for i in ran
ge(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, p in enumerate(self.params):
            x = self.params[i // 2].mm(x) + p.mm(x)
        return x
```

append(parameter)[source]

等价于 `python list` 的 `append` 方法。

参数说明:

- parameter (nn.Parameter) – parameter to append

extend(parameters)[source]

等价于 `python list` 的 `extend` 方法。

参数说明:

- parameters (list) – list of parameters to append

卷积层

class torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)

一维卷积层，输入的尺度是(N, C_in,L)，输出尺度（ N,C_out,L_out）的计算方式：

$$ out(Ni, C{outj})=bias(C {outj})+\sum^{C{in}-1}{k=0}weight(C{out_j},k)\bigotimes input(N_i,k)

$$

说明

`bigotimes`: 表示相关系数计算
`stride`: 控制相关系数的计算步长
`dilation`: 用于控制内核点之间的距离，详细描述在这里
`groups`: 控制输入和输出之间的连接， `group=1` ，输出是所有的输入的卷积； `group=2` ，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

Parameters：

- in_channels(`int`) – 输入信号的通道
- out_channels(`int`) – 卷积产生的通道
- kerner_size(`int` or `tuple`) - 卷积核的尺寸
- stride(`int` or `tuple` , `optional`) - 卷积步长
- padding (`int` or `tuple` , `optional`)- 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional``) – 卷积核元素之间的间距
- groups(`int` , `optional`) – 从输入通道到输出通道的阻塞连接数
- bias(`bool` , `optional`) - 如果 `bias=True` ，添加偏置

shape:
输入: (N,C_in,L_in)
输出: (N,C_out,L_out)
输入输出的计算方式：

$$L{out}=floor((L{in}+2padding-dilation(kernerl_size-1)-1)/stride+1)$$

变量:
weight(`tensor`) - 卷积的权重，大小是(`out_channels` , `in_channels` , `kernel_size`)
bias(`tensor`) - 卷积的偏置系数，大小是（ `out_channel` ）

example:

```
>>> m = nn.Conv1d(16, 33, 3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)

二维卷积层, 输入的尺度是(N, C_in,H,W)，输出尺度（N,C_out,H_out,W_out）的计算方式：

$$out(N*i, C*{out*j})=bias(C*{out*j})+\sum^{C*{in}-1}*{k=0}weight(C*{out_j},k)\bigotimes input(N_i,k)$$

说明

`bigotimes`: 表示二维的相关系数计算 `stride`: 控制相关系数的计算步长
`dilation`: 用于控制内核点之间的距离，详细描述在这里
`groups`: 控制输入和输出之间的连接： `group=1`，输出是所有的输入的卷积；`group=2`，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel_size`，`stride,padding`，`dilation` 也可以是一个 `int` 的数据，此时卷积height和width值相同;也可以是一个 `tuple` 数组，`tuple` 的第一维度表示height的数值，tuple的第二维度表示width的数值

Parameters：

- in_channels(`int`) – 输入信号的通道
- out_channels(`int`) – 卷积产生的通道
- kerner_size(`int` or `tuple`) - 卷积核的尺寸
- stride(`int` or `tuple`, `optional`) - 卷积步长
- padding(`int` or `tuple`, `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple`, `optional`) – 卷积核元素之间的间距
- groups(`int`, `optional`) – 从输入通道到输出通道的阻塞连接数
- bias(`bool`, `optional`) - 如果 `bias=True`，添加偏置

shape:
input: (N,C_in,H_in,W_in)
output: (N,C_out,H_out,W_out)

$$H*{out}=floor((H*{in}+2*padding[0]-dilation[0]*(kernerl_size[0]-1)-1)/stride[0]+1)$$

$$W*{out}=floor((W*{in}+2*padding[1]-dilation[1]*(kernerl_size[1]-1)-1)/stride[1]+1)$$

变量:

weight(`tensor`) - 卷积的权重，大小是(`out_channels` , `in_channels` , `kernel_size`)

bias(`tensor`) - 卷积的偏置系数，大小是（ `out_channel` ）

example:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 100))
>>> output = m(input)
```

class torch.nn.Conv3d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)

三维卷积层, 输入的尺度是(N, C_in,D,H,W)，输出尺度（N,C_out,D_out,H_out,W_out）的计算方式：

$$out(N_i, C_{out_j})=bias(C_{out_j})+\sum^{C_{in}-1}_{k=0}weight(C_{out_j},k)\bigotimes input(N_i,k)$$

说明

`bigotimes` : 表示二维的相关系数计算 `stride` : 控制相关系数的计算步长

`dilation` : 用于控制内核点之间的距离，详细描述在这里

`groups` : 控制输入和输出之间的连接： `group=1` ，输出是所有的输入的卷积； `group=2` ，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel_size` ， `stride` ， `padding` ， `dilation` 可以是一个 `int` 的数据 - 卷积height和width值相同，也可以是一个有三个 `int` 数据的 `tuple` 数组， `tuple` 的第一维度表示depth的数值， `tuple` 的第二维度表示height的数值， `tuple` 的第三维度表示width的数值

Parameters：

- in_channels(`int`) – 输入信号的通道
- out_channels(`int`) – 卷积产生的通道
- kernel_size(`int` or `tuple`) - 卷积核的尺寸
- stride(`int` or `tuple` , `optional`) - 卷积步长
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 卷积核元素之间的间距
- groups(`int` , `optional`) – 从输入通道到输出通道的阻塞连接数
- bias(`bool` , `optional`) - 如果 `bias=True` ，添加偏置

shape:
`input` : (N,C_in,D_in,H_in,W_in)
`output` : (N,C_out,D_out,H_out,W_out)

$$D{out}=floor((D{in}+2padding[0]-dilation[0](kernerl_size[0]-1)-1)/stride[0]+1)$$

$$H{out}=floor((H{in}+2padding[1]-dilation[2](kernerl_size[1]-1)-1)/stride[1]+1)$$

$$W{out}=floor((W{in}+2padding[2]-dilation[2](kernerl_size[2]-1)-1)/stride[2]+1)$$

变量:

- weight(`tensor`) - 卷积的权重，shape是(`out_channels` , `in_channels` , `kernel_size`)`
- bias(`tensor`) - 卷积的偏置系数，shape是（ `out_channel` ）

example:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv3d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(4, 2, 0))
>>> input = autograd.Variable(torch.randn(20, 16, 10, 50, 100))
>>> output = m(input)
```

class torch.nn.ConvTranspose1d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True)

1维的解卷积操作（ `transposed convolution operator` ，注意改视作操作可视作解卷积操作，但并不是真正的解卷积操作） 该模块可以看作是 `Conv1d` 相对于其输入的梯度，有时（但不正确地）被称为解卷积操作。

注意
由于内核的大小，输入的最后的一些列的数据可能会丢失。因为输入和输出是不是完全的互相关。因此，用户可以进行适当的填充（padding操作）。

参数

- in_channels(`int`) – 输入信号的通道数
- out_channels(`int`) – 卷积产生的通道
- kernel_size(`int` or `tuple`) - 卷积核的大小
- stride(`int` or `tuple` , `optional`) - 卷积步长
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- output_padding(`int` or `tuple` , `optional`) - 输出的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 卷积核元素之间的间距
- groups(`int` , `optional`) – 从输入通道到输出通道的阻塞连接数

- bias(`bool` , `optional`) - 如果 `bias=True` ，添加偏置

shape:
输入: (N,C_in,L_in)
输出: (N,C_out,L_out)

$$L*{out}=(L*{in}-1)*stride-2*padding+kernel_size+output_padding$$

变量:

- weight(`tensor`) - 卷积的权重，大小是(`in_channels` , `in_channels` , `kernel_size`)
- bias(`tensor`) - 卷积的偏置系数，大小是(`out_channel`)

class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True)

2维的转置卷积操作（ `transposed convolution operator` ，注意改视作操作可视作解卷积操作，但并不是真正的解卷积操作） 该模块可以看作是 `Conv2d` 相对于其输入的梯度，有时（但不正确地）被称为解卷积操作。

说明

`stride` : 控制相关系数的计算步长
`dilation` : 用于控制内核点之间的距离，详细描述在这里
`groups` : 控制输入和输出之间的连接： `group=1` ，输出是所有的输入的卷积； `group=2` ，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel_size` ， `stride` ， `padding` ， `dilation` 数据类型： 可以是一个 `int` 类型的数据，此时卷积height和width值相同; 也可以是一个 `tuple` 数组（包含来两个 `int` 类型的数据），第一个 `int` 数据表示 `height` 的数值，第二个 `int` 类型的数据表示width的数值

注意
由于内核的大小，输入的最后的一些列的数据可能会丢失。因为输入和输出是不是完全的互相关。因此，用户可以进行适当的填充（ `padding` 操作）。

参数：

- in_channels(`int`) – 输入信号的通道数
- out_channels(`int`) – 卷积产生的通道数
- kerner_size(`int` or `tuple`) - 卷积核的大小
- stride(`int` or `tuple` , `optional`) - 卷积步长
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- output_padding(`int` or `tuple` , `optional`) - 输出的每一条边补充0的层数

- dilation(`int` or `tuple`, `optional`) – 卷积核元素之间的间距
- groups(`int`, `optional`) – 从输入通道到输出通道的阻塞连接数
- bias(`bool`, `optional`) - 如果 `bias=True`，添加偏置

shape:
输入: (N,C_in,H_in，W_in)
输出: (N,C_out,H_out,W_out)

$$H{out}=(H{in}-1)stride[0]-2padding[0]+kernel_size[0]+output_padding[0]$$

$$W{out}=(W{in}-1)stride[1]-2padding[1]+kernel_size[1]+output_padding[1]$$

变量:

- weight(`tensor`) - 卷积的权重，大小是(`in_channels`, `in_channels`, `kernel_size`)
- bias(`tensor`) - 卷积的偏置系数，大小是（ `out_channel` ）

Example

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 100))
>>> output = m(input)
>>> # exact output size can be also specified as an argument
>>> input = autograd.Variable(torch.randn(1, 16, 12, 12))
>>> downsample = nn.Conv2d(16, 16, 3, stride=2, padding=1)
>>> upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
>>> h = downsample(input)
>>> h.size()
torch.Size([1, 16, 6, 6])
>>> output = upsample(h, output_size=input.size())
>>> output.size()
torch.Size([1, 16, 12, 12])
```

torch.nn.ConvTranspose3d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True)

3维的转置卷积操作（ `transposed convolution operator` ，注意改视作操作可视作解卷积操作，但并不是真正的解卷积操作） 转置卷积操作将每个输入值和一个可学习权重的卷积核相乘，输出所有输入通道的求和

该模块可以看作是 `Conv3d` 相对于其输入的梯度，有时（但不正确地）被称为解卷积操作。

说明

`stride` : 控制相关系数的计算步长
`dilation` : 用于控制内核点之间的距离，详细描述在这里
`groups` : 控制输入和输出之间的连接： `group=1` ，输出是所有的输入的卷积； `group=2` ，此

时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel\_size` ，`stride`，`padding`，`dilation` 数据类型：一个 `int` 类型的数据，此时卷积height和width值相同; 也可以是一个 `tuple` 数组（包含来两个 `int` 类型的数据），第一个 `int` 数据表示height的数值，tuple的第二个int类型的数据表示width的数值

注意
由于内核的大小，输入的最后的一些列的数据可能会丢失。因为输入和输出是不是完全的互相关。因此，用户可以进行适当的填充（padding操作）。

参数：

- in_channels(`int`) – 输入信号的通道数
- out_channels(`int`) – 卷积产生的通道数
- kernel_size(`int` or `tuple`) - 卷积核的大小
- stride(`int` or `tuple` , `optional`) - 卷积步长
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- output_padding(`int` or `tuple` , `optional`) - 输出的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 卷积核元素之间的间距
- groups(`int` , `optional`) – 从输入通道到输出通道的阻塞连接数
- bias(`bool` , `optional`) - 如果 `bias=True` ，添加偏置

shape:
输入: (N,C_in,H_in，W_in)
输出: (N,C_out,H_out,W_out)

$$D_{out}=(D_{in}-1)stride[0]-2padding[0]+kernel_size[0]+output_padding[0]$$

$$H_{out}=(H_{in}-1)stride[1]-2padding[1]+kernel_size[1]+output_padding[0]$$

$$W_{out}=(W_{in}-1)stride[2]-2padding[2]+kernel_size[2]+output_padding[2]$$

变量:

- weight(`tensor`) - 卷积的权重，大小是(`in_channels` , `in_channels` , `kernel_size`)
- bias(`tensor`) - 卷积的偏置系数，大小是（ `out_channel` ）

Example

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose3d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(0, 4, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 10, 50, 100))
>>> output = m(input)
```

池化层

class torch.nn.MaxPool1d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)

对于输入信号的输入通道，提供1维最大池化（`max pooling`）操作

如果输入的大小是(N,C,L)，那么输出的大小是(N,C,L_out)的计算方式是：

$$out(N_i, C_j,k)=max^{kernel_size-1}_{m=0}input(N_{i},C_j,stride*k+m)$$

如果 `padding` 不是0，会在输入的每一边添加相应数目0
`dilation` 用于控制内核点之间的距离，详细描述在这里

参数：

- kernel_size(`int` or `tuple`) - max pooling的窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size`
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 一个控制窗口中元素步幅的参数
- return_indices - 如果等于 `True` ，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil_mode - 如果等于 `True` ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

shape:
输入: (N,C_in,L_in)
输出: (N,C_out,L_out)

$$L_{out}=floor((L_{in} + 2padding - dilation(kernel_size - 1) - 1)/stride + 1$$

example:

```
>>> # pool of size=3, stride=2
>>> m = nn.MaxPool1d(3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

class torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)

对于输入信号的输入通道，提供2维最大池化（`max pooling`）操作

如果输入的大小是(N,C,H,W)，那么输出的大小是(N,C,H_out,W_out)和池化窗口大小(kH,kW)的关系是：

$$out(Ni, C_j,k)=max^{kH-1}{m=0}max^{kW-1}
{m=0}input(N{i},C_j,stride[0]h+m,stride[1]w+n)$$

如果 `padding` 不是0，会在输入的每一边添加相应数目0
`dilation` 用于控制内核点之间的距离，详细描述在这里

参数 `kernel_size` ，`stride`，`padding` ，`dilation` 数据类型： 可以是一个 `int` 类型的数据，此时卷积height和width值相同; 也可以是一个 `tuple` 数组（包含来两个int类型的数据），第一个 `int` 数据表示height的数值， `tuple` 的第二个int类型的数据表示width的数值

参数：

- kernel_size(`int` or `tuple`) - max pooling的窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size`
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 一个控制窗口中元素步幅的参数
- return_indices - 如果等于 `True` ，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil_mode - 如果等于 `True` ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

shape:
输入: (N,C,H_{in},W_in)
输出: (N,C,H_out,W_out)

$$H{out}=floor((H{in} + 2padding[0] - dilation[0](kernel_size[0] - 1) - 1)/stride[0] + 1$$

$$W{out}=floor((W{in} + 2padding[1] - dilation[1](kernel_size[1] - 1) - 1)/stride[1] + 1$$

example:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool2d((3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

class torch.nn.MaxPool3d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)

对于输入信号的输入通道，提供3维最大池化（max pooling）操作

如果输入的大小是(N,C,D,H,W)，那么输出的大小是(N,C,D,H_out,W_out)和池化窗口大小(kD,kH,kW)的关系是：

$$out(Ni,C_j,d,h,w)=max^{kD-1}{m=0}max^{kH-1}{m=0}max^{kW-1}{m=0}$$

$$input(N_{i},C_j,stride[0]k+d,stride[1]h+m,stride[2]*w+n)$$

如果 `padding` 不是0，会在输入的每一边添加相应数目0
`dilation` 用于控制内核点之间的距离，详细描述在这里

参数 `kernel_size` ， `stride` , `padding` ， `dilation` 数据类型： 可以是 `int` 类型的数据，此时卷积height和width值相同; 也可以是一个 `tuple` 数组（包含来两个 `int` 类型的数据），第一个 `int` 数据表示height的数值， `tuple` 的第二个 `int` 类型的数据表示width的数值

参数：

- kernel_size(`int` or `tuple`) - max pooling的窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是kernel_size
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 一个控制窗口中元素步幅的参数
- return_indices - 如果等于 `True` ，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil_mode - 如果等于 `True` ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

shape:
输入: (N,C,H_in,W_in)
输出: (N,C,H_out,W_out)

$$D{out}=floor((D{in} + 2padding[0] - dilation[0](kernel_size[0] - 1) - 1)/stride[0] + 1)$$

$$H{out}=floor((H{in} + 2padding[1] - dilation[1](kernel_size[0] - 1) - 1)/stride[1] + 1)$$

$$W{out}=floor((W{in} + 2padding[2] - dilation[2](kernel_size[2] - 1) - 1)/stride[2] + 1)$$

example:

```
>>> # pool of square window of size=3, stride=2
>>>m = nn.MaxPool3d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool3d((3, 2, 2), stride=(2, 1, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50,44, 31))
>>> output = m(input)
```

class torch.nn.MaxUnpool1d(kernel_size, stride=None, padding=0)

`Maxpool1d` 的逆过程，不过并不是完全的逆过程，因为在 `maxpool1d` 的过程中，一些最大值的已经丢失。 `MaxUnpool1d` 输入 `MaxPool1d` 的输出，包括最大值的索引，并计算所有 `maxpool1d` 过程中非最大值被设置为零的部分的反向。

注意：
`MaxPool1d` 可以将多个输入大小映射到相同的输出大小。因此，反演过程可能会变得模棱两可。 为了适应这一点，可以在调用中将输出大小（ `output_size` ）作为额外的参数传入。 具体用法，请参阅下面的输入和示例

参数：

- kernel_size(`int` or `tuple`) - max pooling的窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size`
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数

输入：
`input` :需要转换的 `tensor` `indices` ：Maxpool1d的索引号 `output_size` :一个指定输出大小的 `torch.Size`

shape:
`input` : (N,C,H_in)
`output` :(N,C,H_out)

$$H{out}=(H{in}-1)stride[0]-2padding[0]+kernel_size[0]$$
也可以使用 `output_size` 指定输出的大小

Example：

```
>>> pool = nn.MaxPool1d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool1d(2, stride=2)
>>> input = Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7, 8]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices)
    Variable containing:
    (0 ,.,.) =
       0   2   0   4   0   6   0   8
    [torch.FloatTensor of size 1x1x8]

>>> # Example showcasing the use of output_size
>>> input = Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7, 8, 9]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices, output_size=input.size())
    Variable containing:
    (0 ,.,.) =
       0   2   0   4   0   6   0   8   0
    [torch.FloatTensor of size 1x1x9]
>>> unpool(output, indices)
    Variable containing:
    (0 ,.,.) =
       0   2   0   4   0   6   0   8
    [torch.FloatTensor of size 1x1x8]
```

class torch.nn.MaxUnpool2d(kernel_size, stride=None, padding=0)

`Maxpool2d`的逆过程，不过并不是完全的逆过程，因为在maxpool2d的过程中，一些最大值的已经丢失。 `MaxUnpool2d`的输入是`MaxPool2d`的输出，包括最大值的索引，并计算所有`maxpool2d`过程中非最大值被设置为零的部分的反向。

注意：

`MaxPool2d`可以将多个输入大小映射到相同的输出大小。因此，反演过程可能会变得模棱两可。 为了适应这一点，可以在调用中将输出大小（`output_size`）作为额外的参数传入。具体用法，请参阅下面示例

参数：

- kernel_size(`int` or `tuple`) - max pooling的窗口大小
- stride(`int` or `tuple`, `optional`) - max pooling的窗口移动的步长。默认值是`kernel_size`
- padding(`int` or `tuple`, `optional`) - 输入的每一条边补充0的层数

输入：

`input`:需要转换的`tensor`

`indices`：Maxpool1d的索引号

`output_size`:一个指定输出大小的`torch.Size`

大小：

`input`: (N,C,H_in,W_in)

`output`:(N,C,H_out,W_out)

$$H_{out}=(H_{in}-1)stride[0]-2padding[0]+kernel_size[0]$$

$$W_{out}=(W_{in}-1)stride[1]-2padding[1]+kernel_size[1]$$

也可以使用`output_size`指定输出的大小

Example：

```
>>> pool = nn.MaxPool2d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool2d(2, stride=2)
>>> input = Variable(torch.Tensor([[[[ 1,  2,  3,  4],
    ...                             [ 5,  6,  7,  8],
    ...                             [ 9, 10, 11, 12],
    ...                             [13, 14, 15, 16]]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices)
    Variable containing:
    (0 ,0 ,.,.) =
       0   0   0   0
       0   6   0   8
       0   0   0   0
       0  14   0  16
    [torch.FloatTensor of size 1x1x4x4]

>>> # specify a different output size than input size
>>> unpool(output, indices, output_size=torch.Size([1, 1, 5, 5]))
    Variable containing:
    (0 ,0 ,.,.) =
       0   0   0   0   0
       6   0   8   0   0
       0   0   0  14   0
      16   0   0   0   0
       0   0   0   0   0
    [torch.FloatTensor of size 1x1x5x5]
```

class torch.nn.MaxUnpool3d(kernel_size, stride=None, padding=0)

`Maxpool3d` 的逆过程，不过并不是完全的逆过程，因为在 `maxpool3d` 的过程中，一些最大值的已经丢失。 `MaxUnpool3d` 的输入就是 `MaxPool3d` 的输出，包括最大值的索引，并计算所有 `maxpool3d` 过程中非最大值被设置为零的部分的反向。

注意：
`MaxPool3d` 可以将多个输入大小映射到相同的输出大小。因此，反演过程可能会变得模棱两可。为了适应这一点，可以在调用中将输出大小（ `output_size` ）作为额外的参数传入。具体用法，请参阅下面的输入和示例

参数：

- kernel_size(`int` or `tuple`) - Maxpooling窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size`
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数

输入：

`input` :需要转换的 `tensor`

`indices` ： `Maxpool1d` 的索引序数

`output_size` :一个指定输出大小的 `torch.Size`

大小:

`input` : (N,C,D_in,H_in,W_in)

`outpu` t:(N,C,D_out,H_out,W_out)

$$ \begin{aligned} D{out}=(D{in}-1)stride[0]-2padding[0]+kernel_size[0]\ H{out}=(H{in}-1)stride[1]-2padding[0]+kernel_size[1]\ W{out}=(W{in}-1)stride[2]-2padding[2]+kernel_size[2]
\end{aligned}

$$

也可以使用 `output_size` 指定输出的大小

Example：

```
>>> # pool of square window of size=3, stride=2
>>> pool = nn.MaxPool3d(3, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool3d(3, stride=2)
>>> output, indices = pool(Variable(torch.randn(20, 16, 51, 33, 15)))
>>> unpooled_output = unpool(output, indices)
>>> unpooled_output.size()
torch.Size([20, 16, 51, 33, 15])
```

class torch.nn.AvgPool1d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)

对信号的输入通道，提供1维平均池化（average pooling ） 输入信号的大小(N,C,L)，输出大小(N,C,L_out)和池化窗口大小k的关系是：

$$out(Ni,C_j,l)=1/k*\sum^{k}{m=0}input(N{i},C{j},stride*l+m)$$
如果 `padding` 不是0，会在输入的每一边添加相应数目0

参数：

- kernel_size(`int` or `tuple`) - 池化窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size`
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 一个控制窗口中元素步幅的参数
- return_indices - 如果等于 `True` ，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil_mode - 如果等于 `True` ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

大小：

`input` :(N,C,L_in)
`output` :(N,C,L_out)

$$L{out}=floor((L{in}+2*padding-kernel_size)/stride+1)$$

Example:

```
>>> # pool with window of size=3, stride=2
>>> m = nn.AvgPool1d(3, stride=2)
>>> m(Variable(torch.Tensor([[[1,2,3,4,5,6,7]]])))
Variable containing:
    (0 ,.,.) =
    2  4  6
    [torch.FloatTensor of size 1x1x3]
```

class torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)

对信号的输入通道，提供2维的平均池化（average pooling ）
输入信号的大小(N,C,H,W)，输出大小(N,C,H_out,W_out)和池化窗口大小(kH,kW)的关系是：

$$ out(N*i,C_j,h,w)=1/(kHkW)\sum^{kH-1}*{m=0}\sum^{kW-1}*{n=0}input(N*{i},C_{j},stride[0]*h+m,stride[1]*w+n)$$

如果 `padding` 不是0，会在输入的每一边添加相应数目0

参数：

- kernel_size(`int` or `tuple`) - 池化窗口大小
- stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size`
- padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
- dilation(`int` or `tuple` , `optional`) – 一个控制窗口中元素步幅的参数
- ceil_mode - 如果等于 `True` ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作
- count_include_pad - 如果等于 `True` ，计算平均池化时，将包括 `padding` 填充的0

shape：
`input` : (N,C,H_in,W_in)
`output` : (N,C,H_out,W_out)

$$\begin{aligned} H*{out}=floor((H*{in}+2*padding[0]-kernel_size[0])/stride[0]+1)\ W*{out}=floor((W*{in}+2*padding[1]-kernel_size[1])/stride[1]+1) \end{aligned}

$$

Example:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.AvgPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.AvgPool2d((3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

class torch.nn.AvgPool3d(kernel_size, stride=None)

对信号的输入通道，提供3维的平均池化（`average pooling`） 输入信号的大小(N,C,D,H,W)，输出大小(N,C,D_out,H_out,W_out)和池化窗口大小(kD,kH,kW)的关系是：

$$ \begin{aligned} out(Ni,C_j,d,h,w)=1/(kDkHkW)*\sum^{kD-1}{k=0}\sum^{kH-1}{m=0}\sum^{kW-1}{n=0}input(N{i},C{j},stride[0]d+k,stride[1]h+m,stride[2]*w+n) \end{aligned}

$$ 如果 `padding` 不是0，会在输入的每一边添加相应数目0

参数：

- kernel_size(`int` or `tuple`) - 池化窗口大小
- stride(`int` or `tuple` , `optional`) - max `pooling` 的窗口移动的步长。默认值是 `kernel_size`

shape：
输入大小:(N,C,D_in,H_in,W_in)
输出大小:(N,C,D_out,H_out,W_out)

$$\begin{aligned} D{out}=floor((D{in}+2padding[0]-kernel_size[0])/stride[0]+1)\
H{out}=floor((H{in}+2padding[1]-kernel_size[1])/stride[1]+1)\
W{out}=floor((W{in}+2*padding[2]-kernel_size[2])/stride[2]+1)
\end{aligned}

$$

Example:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.AvgPool3d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.AvgPool3d((3, 2, 2), stride=(2, 1, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50,44, 31))
>>> output = m(input)
```

class torch.nn.FractionalMaxPool2d(kernel_size, output_size=None, output_ratio=None, return_indices=False, _random_samples=None)

对输入的信号，提供2维的分数最大化池化操作 分数最大化池化的细节请阅读论文 由目标输出大小确定的随机步长,在$kH*kW$区域进行最大池化操作。输出特征和输入特征的数量相同。

参数：

- kernel_size(`int` or `tuple`) - 最大池化操作时的窗口大小。可以是一个数字（表

示 `K*K` 的窗口），也可以是一个元组（ `kh*kw` ）
- output_size - 输出图像的尺寸。可以使用一个 `tuple` 指定(oH,oW)，也可以使用一个数字oH指定一个oH*oH的输出。
- output_ratio – 将输入图像的大小的百分比指定为输出图片的大小，使用一个范围在(0,1)之间的数字指定
- return_indices - 默认值 `False` ，如果设置为 `True` ，会返回输出的索引，索引对 `nn.MaxUnpool2d` 有用。

Example：

```
>>> # pool of square window of size=3, and target output size 13x12
>>> m = nn.FractionalMaxPool2d(3, output_size=(13, 12))
>>> # pool of square window and target output size being half of input image size
>>> m = nn.FractionalMaxPool2d(3, output_ratio=(0.5, 0.5))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

class torch.nn.LPPool2d(norm_type, kernel_size, stride=None, ceil_mode=False)

对输入信号提供2维的幂平均池化操作。 输出的计算方式：

$$f(x)=pow(sum(X,p),1/p)$$

- 当p为无穷大的时候时，等价于最大池化操作
- 当 `p=1` 时，等价于平均池化操作

参数 `kernel_size` , `stride` 的数据类型：

- `int` ，池化窗口的宽和高相等
- `tuple` 数组（两个数字的），一个元素是池化窗口的高，另一个是宽

参数

- kernel_size: 池化窗口的大小
- stride：池化窗口移动的步长。 `kernel_size` 是默认值
- ceil_mode: `ceil_mode=True` 时，将使用向下取整代替向上取整

shape

- 输入：(N,C,H_in,W_in)
- 输出：(N,C,H*out,W_out)*
  $$\begin{aligned} H{out} = floor((H*{in}+2padding[0]-dilation[0](kernel_size[0]-1)-1)/stride[0]+1)\ W*{out} = floor((W_{in}+2*padding[1]-dilation[1]*(kernel_size[1]-1)-1)/stride[1]+1) \end{aligned} $$

Example:

```
>>> # power-2 pool of square window of size=3, stride=2
>>> m = nn.LPPool2d(2, 3, stride=2)
>>> # pool of non-square window of power 1.2
>>> m = nn.LPPool2d(1.2, (3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

class torch.nn.AdaptiveMaxPool1d(output_size, return_indices=False)

对输入信号，提供1维的自适应最大池化操作 对于任何输入大小的输入，可以将输出尺寸指定为H，但是输入和输出特征的数目不会变化。

参数：

- output_size: 输出信号的尺寸
- return_indices: 如果设置为 `True` ，会返回输出的索引。对 `nn.MaxUnpool1d` 有用，默认值是 `False`

Example：

```
>>> # target output size of 5
>>> m = nn.AdaptiveMaxPool1d(5)
>>> input = autograd.Variable(torch.randn(1, 64, 8))
>>> output = m(input)
```

class torch.nn.AdaptiveMaxPool2d(output_size, return_indices=False)

对输入信号，提供2维的自适应最大池化操作 对于任何输入大小的输入，可以将输出尺寸指定为H*W，但是输入和输出特征的数目不会变化。

参数：

- output_size: 输出信号的尺寸,可以用（H,W）表示 `H*W` 的输出，也可以使用数字 `H` 表示 `H*H` 大小的输出
- return_indices: 如果设置为 `True` ，会返回输出的索引。对 `nn.MaxUnpool2d` 有用，默认值是 `False`

Example：

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveMaxPool2d((5,7))
>>> input = autograd.Variable(torch.randn(1, 64, 8, 9))
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveMaxPool2d(7)
>>> input = autograd.Variable(torch.randn(1, 64, 10, 9))
>>> output = m(input)
```

class torch.nn.AdaptiveAvgPool1d(output_size)

对输入信号，提供1维的自适应平均池化操作 对于任何输入大小的输入，可以将输出尺寸指定为H*W，但是输入和输出特征的数目不会变化。

参数：

- output_size: 输出信号的尺寸

Example：

```
>>> # target output size of 5
>>> m = nn.AdaptiveAvgPool1d(5)
>>> input = autograd.Variable(torch.randn(1, 64, 8))
>>> output = m(input)
```

class torch.nn.AdaptiveAvgPool2d(output_size)

对输入信号，提供2维的自适应平均池化操作 对于任何输入大小的输入，可以将输出尺寸指定为 `H*W` ，但是输入和输出特征的数目不会变化。

参数：

- output_size: 输出信号的尺寸,可以用(H,W)表示 `H*W` 的输出，也可以使用耽搁数字H表示H*H大小的输出

Example：

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveAvgPool2d((5,7))
>>> input = autograd.Variable(torch.randn(1, 64, 8, 9))
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveAvgPool2d(7)
>>> input = autograd.Variable(torch.randn(1, 64, 10, 9))
>>> output = m(input)
```

Non-Linear Activations [source]

> class torch.nn.ReLU(inplace=False) [source]

对输入运用修正线性单元函数${ReLU}(x)= max(0, x)$，

参数： inplace-选择是否进行覆盖运算

shape：

- 输入：$(N,)$，代表任意数目附加维度
- 输出：$(N, *)$，与输入拥有同样的shape属性

例子：

```
>>> m = nn.ReLU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.ReLU6(inplace=False) [source]

对输入的每一个元素运用函数${ReLU6}(x) = min(max(0,x), 6)$，

参数： inplace-选择是否进行覆盖运算

shape：

- 输入：$(N,)$，代表任意数目附加维度
- 输出：$(N, *)$，与输入拥有同样的shape属性

例子：

```
>>> m = nn.ReLU6()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.ELU(alpha=1.0, inplace=False) [source]

对输入的每一个元素运用函数$f(x) = max(0,x) + min(0, alpha * (e^x - 1))$，

shape：

- 输入：$(N, *)$，星号代表任意数目附加维度
- 输出：$(N, *)$与输入拥有同样的shape属性

例子：

```
>>> m = nn.ELU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.PReLU(num_parameters=1, init=0.25)[source]

对输入的每一个元素运用函数$PReLU(x) = max(0,x) + a * min(0,x)$，`a` 是一个可学习参数。当没有声明时，`nn.PReLU()` 在所有的输入中只有一个参数 `a` ；如果是 `nn.PReLU(nChannels)` ，`a` 将应用到每个输入。

注意：当为了表现更佳的模型而学习参数 `a` 时不要使用权重衰减（weight decay）

参数：

- num_parameters：需要学习的 `a` 的个数，默认等于1
- init： `a` 的初始值，默认等于0.25

shape：

- 输入：$(N,)$，代表任意数目附加维度
- 输出：$(N, *)$，与输入拥有同样的shape属性

例子：

```
>>> m = nn.PReLU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.LeakyReLU(negative_slope=0.01, inplace=False) [source]

对输入的每一个元素运用$f(x) = max(0, x) + {negative_slope} * min(0, x)$

参数：

- negative_slope：控制负斜率的角度，默认等于0.01
- inplace-选择是否进行覆盖运算

shape：

- 输入：$(N,)$，代表任意数目附加维度
- 输出：$(N, *)$，与输入拥有同样的shape属性

例子：

```
>>> m = nn.LeakyReLU(0.1)
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Threshold(threshold, value, inplace=False) [source]

Threshold定义：

$$ y = x ,if\ x >= threshold\ y = value,if\ x < threshold

$$

参数：

- threshold：阈值
- value：输入值小于阈值则会被value代替
- inplace：选择是否进行覆盖运算

shape：

- 输入：$(N,)$，代表任意数目附加维度
- 输出：$(N, *)$，与输入拥有同样的shape属性

例子：

```
>>> m = nn.Threshold(0.1, 20)
>>> input = Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Hardtanh(min_value=-1, max_value=1, inplace=False) [source]

对每个元素，

$$ f(x) = +1, if\ x > 1;\ f(x) = -1, if\ x < -1;\ f(x) = x, otherwise

$$

线性区域的范围[-1,1]可以被调整

参数：

- min_value：线性区域范围最小值
- max_value：线性区域范围最大值
- inplace：选择是否进行覆盖运算

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Hardtanh()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Sigmoid [source]

对每个元素运用Sigmoid函数，Sigmoid 定义如下：

$$f(x) = 1 / (1 + e^{-x})$$

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Sigmoid()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

class torch.nn.Tanh [source]

对输入的每个元素，

$$f(x) = \frac{e^{x} - e^{-x}} {e^{x} + e^{x}}$$

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Tanh()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

class torch.nn.LogSigmoid [source]

对输入的每个元素，$LogSigmoid(x) = log(1 / (1 + e^{-x}))$

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.LogSigmoid()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

class torch.nn.Softplus(beta=1, threshold=20)[source]

对每个元素运用Softplus函数，Softplus 定义如下：

$$f(x) = \frac{1}{beta} log(1 + e^{(beta x_i)})$$

Softplus函数是ReLU函数的平滑逼近，Softplus函数可以使得输出值限定为正数。

为了保证数值稳定性，线性函数的转换可以使输出大于某个值。

参数：

- beta：Softplus函数的beta值
- threshold：阈值

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Softplus()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Softshrink(lambd=0.5)[source]

对每个元素运用Softshrink函数，Softshrink函数定义如下：

$$ f(x) = x-lambda, if\ x > lambda\ f(x) = x+lambda, if\ x < -lambda\ f(x) = 0, otherwise
$$

参数：

lambd：Softshrink函数的lambda值，默认为0.5

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Softshrink()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Softsign [source]

$f(x) = x / (1 + |x|)$

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Softsign()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Softshrink(lambd=0.5)[source]

对每个元素运用Tanhshrink函数，Tanhshrink函数定义如下：

$$ Tanhshrink(x) = x - Tanh(x)
$$

shape：

- 输入：(N, *)，*表示任意维度组合
- 输出：(N, *)，与输入有相同的shape属性

例子：

```
>>> m = nn.Tanhshrink()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Softmin [source]

对n维输入张量运用Softmin函数，将张量的每个元素缩放到（0,1）区间且和为1。Softmin函数定义如下：

$$f_i(x) = \frac{e^{(-x_i - shift)}} { \sum^j e^{(-x_j - shift)}},shift = max (x_i)$$

shape：

- 输入：(N, L)
- 输出：(N, L)

例子：

```
>>> m = nn.Softmin()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.Softmax [source]

对n维输入张量运用Softmax函数，将张量的每个元素缩放到（0,1）区间且和为1。Softmax函数定义如下：

$$f_i(x) = \frac{e^{(x_i - shift)}} { \sum^j e^{(x_j - shift)}},shift = max (x_i)$$

shape：

- 输入：(N, L)
- 输出：(N, L)

返回结果是一个与输入维度相同的张量，每个元素的取值范围在（0,1）区间。

例子：

```
>>> m = nn.Softmax()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

> class torch.nn.LogSoftmax [source]

对n维输入张量运用LogSoftmax函数，LogSoftmax函数定义如下：

$$f_i(x) = log \frac{e^{(x_i)}} {a}, a = \sum^j e^{(x_j)}$$

shape：

- 输入：(N, L)
- 输出：(N, L)

例子：

```
>>> m = nn.LogSoftmax()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

Normalization layers [source]

class torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True) [source]

对小批量(mini-batch)的2d或3d输入进行批标准化(Batch Normalization)操作

$$ y = \frac{x - mean[x]}{ \sqrt{Var[x]} + \epsilon} * gamma + beta $$

在每一个小批量（mini-batch）数据中，计算输入各个维度的均值和标准差。gamma与beta是可学习的大小为C的参数向量（C为输入大小）

在训练时，该层计算每次输入的均值与方差，并进行移动平均。移动平均默认的动量值为0.1。

在验证时，训练求得的均值/方差将用于标准化验证数据。

参数：

- **num_features**： 来自期望输入的特征数，该期望输入的大小为'batch_size x num_features [x width]'
- **eps**： 为保证数值稳定性（分母不能趋近或取0）,给分母加上的值。默认为1e-5。
- **momentum**： 动态均值和动态方差所使用的动量。默认为0.1。
- **affine**： 一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

Shape：

- 输入：（N, C）或者(N, C, L)
- 输出：（N, C）或者（N，C，L）（输入输出相同）

例子

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm1d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm1d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100))
>>> output = m(input)
```

---

class torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)[source]

对小批量(mini-batch)3d数据组成的4d输入进行批标准化(Batch Normalization)操作

$$ y = \frac{x - mean[x]}{ \sqrt{Var[x]} + \epsilon} * gamma + beta $$

在每一个小批量（mini-batch）数据中，计算输入各个维度的均值和标准差。gamma与beta是可学习的大小为C的参数向量（C为输入大小）

在训练时，该层计算每次输入的均值与方差，并进行移动平均。移动平均默认的动量值为0.1。

在验证时，训练求得的均值/方差将用于标准化验证数据。

参数：

- **num_features**： 来自期望输入的特征数，该期望输入的大小为'batch_size x num_features x height x width'
- **eps**： 为保证数值稳定性（分母不能趋近或取0）,给分母加上的值。默认为1e-5。
- **momentum**： 动态均值和动态方差所使用的动量。默认为0.1。
- **affine**： 一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

Shape：

- 输入：（N, C，H, W)
- 输出：（N, C, H, W）（输入输出相同）

例子

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm2d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100, 35, 45))
>>> output = m(input)
```

---

class torch.nn.BatchNorm3d(num_features, eps=1e-05, momentum=0.1, affine=True)[source]

对小批量(mini-batch)4d数据组成的5d输入进行批标准化(Batch Normalization)操作

$$ y = \frac{x - mean[x]}{ \sqrt{Var[x]} + \epsilon} * gamma + beta $$

在每一个小批量（mini-batch）数据中，计算输入各个维度的均值和标准差。gamma与beta是可学习的大小为C的参数向量（C为输入大小）

在训练时，该层计算每次输入的均值与方差，并进行移动平均。移动平均默认的动量值为0.1。

在验证时，训练求得的均值/方差将用于标准化验证数据。

参数：

- **num_features**： 来自期望输入的特征数，该期望输入的大小为'batch_size x num_features depth x height x width'
- **eps**： 为保证数值稳定性（分母不能趋近或取0）,给分母加上的值。默认为1e-5。
- **momentum**： 动态均值和动态方差所使用的动量。默认为0.1。
- **affine**： 一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

Shape：

- 输入：（N, C，D, H, W)
- 输出：（N, C, D, H, W）（输入输出相同）

例子

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm3d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm3d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100, 35, 45, 10))
>>> output = m(input)
```

Recurrent layers

class torch.nn.RNN(*args,* * kwargs)[source]

将一个多层的 `Elman RNN` ，激活函数为 `tanh` 或者 `ReLU` ，用于输入序列。

对输入序列中每个元素， `RNN` 每层的计算公式为

$$ h*t=tanh(w*{ih} x*t+b*{ih}+w_*{hh} h*{t-1}+b*{hh})

$$ h_t是时刻t的隐状态。 x_t是上一层时刻t的隐状态，或者是第一层在时刻t的输入。如果 `nonlinearity='relu'` ,那么将使用 `relu` 代替 `tanh` 作为激活函数。

参数说明:

- input_size – 输入 `x` 的特征数量。
- hidden_size – 隐层的特征数量。
- num_layers – RNN的层数。
- nonlinearity – 指定非线性函数使用 `tanh` 还是 `relu` 。默认是 `tanh` 。
- bias – 如果是 `False` ，那么RNN层就不会使用偏置权重 b_ih和b_hh,默认是 `True`
- batch_first – 如果 `True` 的话，那么输入 `Tensor` 的shape应该是[batch_size, time_step, feature],输出也是这样。
- dropout – 如果值非零，那么除了最后一层外，其它层的输出都会套上一个 `dropout` 层。
- bidirectional – 如果 `True` ，将会变成一个双向 `RNN` ，默认为 `False` 。

`RNN` 的输入： **(input, h_0)**

- input (seq_len, batch, input_size): 保存输入序列特征的 `tensor` 。 `input` 可以是被填充的变长的序列。细节请看 `torch.nn.utils.rnn.pack_padded_sequence()`
- h_0 (num_layers * num_directions, batch, hidden_size): 保存着初始隐状态的 `tensor`

`RNN` 的输出： **(output, h_n)**

- output (seq_len, batch, hidden_size * num_directions): 保存着 `RNN` 最后一层的输出特征。如果输入是被填充过的序列，那么输出也是被填充的序列。
- h_n (num_layers * num_directions, batch, hidden_size): 保存着最后一个时刻隐状态。

`RNN` 模型参数:

- weight_ih_l[k] – 第 `k` 层的 `input-hidden` 权重， 可学习，形状是 `(input_size x hidden_size)` 。
- weight_hh_l[k] – 第 `k` 层的 `hidden-hidden` 权重， 可学习，形状是 `(hidden_size x hidden_size)`
- bias_ih_l[k] – 第 `k` 层的 `input-hidden` 偏置， 可学习，形状是 `(hidden_size)`
- bias_hh_l[k] – 第 `k` 层的 `hidden-hidden` 偏置， 可学习，形状是 `(hidden_size)`

示例：

```
rnn = nn.RNN(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
output, hn = rnn(input, h0)
```

class torch.nn.LSTM(*args,* * kwargs)[source]

将一个多层的 `(LSTM)` 应用到输入序列。

对输入序列的每个元素， `LSTM` 的每层都会执行以下计算：

$$ \begin{aligned} it &= *sigmoid(W*{ii}xt+*b*{ii}+W*{hi}h*{t-1}+b*{hi}) \ f_t &= sigmoid(W*{if}xt+*b*{if}+W*{hf}h*{t-1}+b*{hf}) \ o_t &= sigmoid(W*{io}xt+*b*{io}+W*{ho}h*{t-1}+b*{ho})\ g_t &= tanh(W*{ig}xt+*b*{ig}+W*{hg}h*{t-1}+b*{hg})\ c_t &= f_t*c*{t-1}+i_tg_t\ *h_t* &= *o_t*tanh(c_t) \end{aligned}

$$ h_t是时刻t的隐状态,c_t是时刻t的细胞状态，x_t是上一层的在时刻t的隐状态或者是第一层在时刻t的输入。i_t, f_t, g_t, o_t 分别代表 输入门，遗忘门，细胞和输出门。

参数说明:

- input_size – 输入的特征维度
- hidden_size – 隐状态的特征维度
- num_layers – 层数（和时序展开要区分开）
- bias – 如果为 `False` ，那么 `LSTM` 将不会使用$b*{ih},b*{hh}$，默认为 `True` 。
- batch_first – 如果为 `True` ，那么输入和输出 `Tensor` 的形状为 `(batch, seq, feature)`

- dropout – 如果非零的话，将会在 `RNN` 的输出上加个 `dropout` ，最后一层除外。
- bidirectional – 如果为 `True` ，将会变成一个双向 `RNN` ，默认为 `False` 。

`LSTM` 输入: input, (h_0, c_0)

- input (seq_len, batch, input_size): 包含输入序列特征的 `Tensor` 。也可以是 `packed variable` ，详见 pack_padded_sequence
- h_0 (num_layers * num_directions, batch, hidden_size):保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`
- c_0 (num_layers * num_directions, batch, hidden_size): 保存着 `batch` 中每个元素的初始化细胞状态的 `Tensor`

`LSTM` 输出 output, (h_n, c_n)

- output (seq_len, batch, hidden_size * num_directions): 保存 `RNN` 最后一层的输出的 `Tensor` 。 如果输入是 `torch.nn.utils.rnn.PackedSequence` ，那么输出也是 `torch.nn.utils.rnn.PackedSequence` 。
- h_n (num_layers * num_directions, batch, hidden_size): `Tensor` ，保存着 `RNN` 最后一个时间步的隐状态。
- c_n (num_layers * num_directions, batch, hidden_size): `Tensor` ，保存着 `RNN` 最后一个时间步的细胞状态。

`LSTM` 模型参数:

- weight*ih_l[k]* – 第 `k` 层可学习的 *`input-hidden`* 权重*($W*{ii}|W*{if}*|*W*{ig}|W_{io}$)，形状为 `(input_size x 4*hidden_size)`
- weight*hh_l[k]* – 第 `k` 层可学习的 *`hidden-hidden`* 权重*($W*{hi}|W*{hf}*|*W*{hg}|W_{ho}$)，形状为 `(hidden_size x 4*hidden_size)` 。
- bias*ih_l[k]* – 第 `k` 层可学习的 *`input-hidden`* 偏置*($b*{ii}|b*{if}*|*b*{ig}|b_{io}$)，形状为 `( 4*hidden_size)`
- bias*hh_l[k]* – 第 `k` 层可学习的 *`hidden-hidden`* 偏置*($b*{hi}|b*{hf}*|*b*{hg}|b_{ho}$)，形状为 `( 4*hidden_size)` 。 示例:

```
lstm = nn.LSTM(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
c0 = Variable(torch.randn(2, 3, 20))
output, hn = lstm(input, (h0, c0))
```

class torch.nn.GRU(*args,* * kwargs)[source]

将一个多层的 `GRU` 用于输入序列。

对输入序列中的每个元素，每层进行了一下计算：

$$ \begin{aligned} rt&=sigmoid(W{ir}xt+b{ir}+W{hr}h{(t-1)}+b{hr})\ i_t&=sigmoid(W{ii}xt+b{ii}+W{hi}h{(t-1)}+b{hi})\ n_t&=tanh(W{in}xt+b{in}+rt(W{hn}h{(t-1)}+b_{hn}))\ h_t&=(1-i_t) nt+i_t*h(t-1) \end{aligned}

$$ h_t是是时间t的上的隐状态，x_t是前一层t时刻的隐状态或者是第一层的t时刻的输入，r_t, i_t, n_t分别是重置门，输入门和新门。

参数说明：

- input_size – 期望的输入x的特征值的维度
- hidden_size – 隐状态的维度
- num_layers – `RNN` 的层数。
- bias – 如果为 `False` ，那么 `RNN` 层将不会使用 `bias` ，默认为 `True` 。
- batch_first – 如果为 `True` 的话，那么输入和输出的 `tensor` 的形状是 `(batch, seq, feature)` 。
- dropout – 如果非零的话，将会在 `RNN` 的输出上加个 `dropout` ，最后一层除外。
- bidirectional – 如果为 `True` ，将会变成一个双向 `RNN` ，默认为 `False` 。

输入： input, h_0

- input (seq_len, batch, input_size): 包含输入序列特征的 `Tensor` 。也可以是 `packed variable` ，详见 pack_padded_sequence 。
- h_0 (num_layers * num_directions, batch, hidden_size):保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`

输出： output, h_n

- output (seq_len, batch, hidden_size * num_directions): ten保存 `RNN` 最后一层的输出的 `Tensor` 。 如果输入是 `torch.nn.utils.rnn.PackedSequence` ，那么输出也是 `torch.nn.utils.rnn.PackedSequence` 。
- h_n (num_layers * num_directions, batch, hidden_size): `Tensor` ，保存着 `RNN` 最后一个时间步的隐状态。

变量：

- weightih_l[k] – 第 `k` 层可学习的 `input-hidden` 权重($W{ir}|W{ii}|W{in}$)，形状为 `(input_size x 3*hidden_size)`
- weighthh_l[k] – 第 `k` 层可学习的 `hidden-hidden` 权重($W{hr}|W{hi}|W{hn}$)，形状为 `(hidden_size x 3*hidden_size)` 。

- biasih_l[k] – 第 `k` 层可学习的 `input-hidden` 偏置(b{ir}|b{ii}|b{in}$)，形状为 `( 3*hidden_size)`

- biashh_l[k] – 第 `k` 层可学习的 `hidden-hidden` 偏置(b{hr}|b{hi}|b{hn}$)，形状为 `( 3*hidden_size)` 。

例子：

```
rnn = nn.GRU(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
output, hn = rnn(input, h0)
```

class torch.nn.RNNCell(input_size, hidden_size, bias=True, nonlinearity='tanh')[source]

一个 `Elan RNN cell` ，激活函数是 `tanh` 或 `ReLU` ，用于输入序列。 将一个多层的 `Elman RNNCell` ，激活函数为 `tanh` 或者 `ReLU` ，用于输入序列。

$$ h'=tanh(w{ih}* x+b{ih}+w{hh}* h+b{hh})

$$ 如果 `nonlinearity=relu` ，那么将会使用 `ReLU` 来代替 `tanh` 。

参数：

- input_size – 输入x，特征的维度。
- hidden_size – 隐状态特征的维度。
- bias – 如果为 `False` ， `RNN cell` 中将不会加入 `bias` ，默认为 `True` 。
- nonlinearity – 用于选择非线性激活函数 [`tanh` | `relu`]. 默认值为： `tanh`

输入： input, hidden

- input (batch, input_size): 包含输入特征的 `tensor` 。
- hidden (batch, hidden_size): 保存着初始隐状态值的 `tensor` 。

输出： h’

- h’ (batch, hidden_size):下一个时刻的隐状态。

变量：

- weight_ih – `input-hidden` 权重， 可学习，形状是 `(input_size x hidden_size)` 。
- weight_hh – `hidden-hidden` 权重， 可学习，形状是 `(hidden_size x hidden_size)`
- bias_ih – `input-hidden` 偏置， 可学习，形状是 `(hidden_size)`

- bias_hh – `hidden-hidden` 偏置， 可学习，形状是 `(hidden_size)`

例子：

```
rnn = nn.RNNCell(10, 20)
input = Variable(torch.randn(6, 3, 10))
hx = Variable(torch.randn(3, 20))
output = []
for i in range(6):
    hx = rnn(input[i], hx)
    output.append(hx)
```

class torch.nn.LSTMCell(input_size, hidden_size, bias=True)[source]

`LSTM cell` 。

$$ \begin{aligned} i &= sigmoid(W*{ii}x+b*{ii}+W*{hi}h+b*{hi}) \ f &= sigmoid(W*{if}x+b*{if}+W*{hf}h+b*{hf}) \ o &= sigmoid(W*{io}x+b*{io}+W*{ho}h+b*{ho})\ g &= tanh(W*{ig}x+b*{ig}+W*{hg}h+b*{hg})\ c' &= f*t*c*{t-1}+i_t*g_t\ h' &= o_t*tanh(c') \end{aligned}

$$

参数：

- input_size – 输入的特征维度。
- hidden_size – 隐状态的维度。
- bias – 如果为 `False` ，那么将不会使用 `bias` 。默认为 `True` 。

`LSTM` 输入: input, (h_0, c_0)

- input (seq_len, batch, input_size): 包含输入序列特征的 `Tensor` 。也可以是 `packed variable` ，详见 pack_padded_sequence

- h_0 (batch, hidden_size):保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`

- c_0 (batch, hidden_size): 保存着 `batch` 中每个元素的初始化细胞状态的 `Tensor`

输出： h_1, c_1

- h_1 (batch, hidden_size): 下一个时刻的隐状态。
- c_1 (batch, hidden_size): 下一个时刻的细胞状态。

`LSTM` 模型参数:

- weight*ih* – *`input-hidden`* 权重*($W*{ii}|W*{if}|W*{ig}|W_{io}$)，形状为 `(input_size x 4*hidden_size)`

- weight*hh* – `hidden-hidden` 权重($W{hi}|W{hf}|W{hg}|W_{ho}$)，形状为 `(hidden_size x 4*hidden_size)` 。
- bias*ih* – `input-hidden` 偏置($b{ii}|b{if}|b{ig}|b_{io}$)，形状为 `( 4*hidden_size)`
- bias*hh* – `hidden-hidden` 偏置($b{hi}|b{hf}|b{hg}|b_{ho}$)，形状为 `( 4*hidden_size)` 。

Examples:

```
rnn = nn.LSTMCell(10, 20)
input = Variable(torch.randn(6, 3, 10))
hx = Variable(torch.randn(3, 20))
cx = Variable(torch.randn(3, 20))
output = []
for i in range(6):
   hx, cx = rnn(input[i], (hx, cx))
   output.append(hx)
```

class torch.nn.GRUCell(input_size, hidden_size, bias=True) [source]

一个 `GRU cell` 。

$$ \begin{aligned} r&=sigmoid(W{ir}x+b{ir}+W{hr}h+b{hr})\ i&=sigmoid(W{ii}x+b{ii}+W{hi}h+b{hi})\ n&=tanh(W{in}x+b{in}+r(W{hn}h+b{hn}))\ h'&=(1-i) n+i*h \end{aligned}

$$

参数说明：

- input_size – 期望的输入x的特征值的维度
- hidden_size – 隐状态的维度
- bias – 如果为 `False` ，那么 `RNN` 层将不会使用 `bias` ，默认为 `True` 。

输入： input, h_0

- input (batch, input_size): 包含输入特征的 `Tensor`
- h_0 (batch, hidden_size):保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`

输出： h_1

- h_1 (batch, hidden_size): `Tensor` ，保存着 `RNN` 下一个时刻的隐状态。

变量：

- weight*ih* – `input-hidden` 权重($W{ir}|W{ii}|W{in}$)，形状为 `(input_size x 3*hidden_size)`

- weight*hh* – `hidden-hidden` 权重(*$W*{hr}|W*{hi}*|*W*{hn}$)，形状为 `(hidden_size x 3*hidden_size)` 。
- bias*ih* – `input-hidden` 偏置(*$b*{ir}|b*{ii}*|*b*{in}$)，形状为 `( 3*hidden_size)`
- bias*hh* – `hidden-hidden` 偏置(*$b*{hr}|b*{hi}*|*b*{hn}$)，形状为 `( 3*hidden_size)` 。

例子：

```
rnn = nn.GRUCell(10, 20)
input = Variable(torch.randn(6, 3, 10))
hx = Variable(torch.randn(3, 20))
output = []
for i in range(6):
   hx = rnn(input[i], hx)
   output.append(hx)
```

Linear layers

```
class torch.nn.Linear(in_features, out_features, bias=True)
```

对输入数据做线性变换：\(y = Ax + b\)

参数：

- **in_features** - 每个输入样本的大小
- **out_features** - 每个输出样本的大小
- **bias** - 若设置为False，这层不会学习偏置。默认值：True

形状：

- 输入**:** \((N, in_features)\)
- 输出： \((N, out_features)\)

变量：

- **weight** -形状为(out_features x in_features)的模块中可学习的权值
- **bias** -形状为(out_features)的模块中可学习的偏置

例子：

```
>>> m = nn.Linear(20, 30)
>>> input = autograd.Variable(torch.randn(128, 20))
>>> output = m(input)
>>> print(output.size())
```

Dropout layers

```
class torch.nn.Dropout(p=0.5, inplace=False)
```

随机将输入张量中部分元素设置为0。对于每次前向调用，被置0的元素都是随机的。

参数：

- **p** - 将元素置0的概率。默认值：0.5
- **in-place** - 若设置为True，会在原地执行操作。默认值：False

形状：

- 输入： 任意。输入可以为任意形状。
- 输出： 相同。输出和输入形状相同。

例子：

```
>>> m = nn.Dropout(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16))
>>> output = m(input)
```

```
class torch.nn.Dropout2d(p=0.5, inplace=False)
```

随机将输入张量中整个通道设置为0。对于每次前向调用，被置0的通道都是随机的。

通常输入来自*Conv2d*模块。

像在论文Efficient Object Localization Using Convolutional Networks，如果特征图中相邻像素是强相关的（在前几层卷积层很常见），那么iid dropout不会归一化激活，而只会降低学习率。

在这种情形，`nn.Dropout2d()`可以提高特征图之间的独立程度，所以应该使用它。

参数：

- **p**(*float*, *optional*) - 将元素置0的概率。
- **in-place**(*bool*, *optional*) - 若设置为True，会在原地执行操作。

形状：

- 输入： \((N, C, H, W)\)
- 输出： \((N, C, H, W)\)（与输入形状相同）

例子：

```
>>> m = nn.Dropout2d(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16, 32, 32))
>>> output = m(input)
```

```
class torch.nn.Dropout3d(p=0.5, inplace=False)
```

随机将输入张量中整个通道设置为0。对于每次前向调用，被置0的通道都是随机的。

通常输入来自*Conv3d*模块。

像在论文Efficient Object Localization Using Convolutional Networks，如果特征图中相邻像素是强相关的（在前几层卷积层很常见），那么iid dropout不会归一化激活，而只会降低学习率。

在这种情形，`nn.Dropout3d()` 可以提高特征图之间的独立程度，所以应该使用它。

参数：

- **p**(*float, optional*) - 将元素置0的概率。
- **in-place**(*bool, optional*) - 若设置为True，会在原地执行操作。

形状：

- 输入：\(N, C, D, H, W)\)
- 输出：\((N, C, D, H, W)\)（与输入形状相同）

例子：

```
>>> m = nn.Dropout3d(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16, 4, 32, 32))
>>> output = m(input)
```

Sparse layers

```
class torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=Non
e, norm_type=2, scale_grad_by_freq=False, sparse=False)
```

一个保存了固定字典和大小的简单查找表。

这个模块常用来保存词嵌入和用下标检索它们。模块的输入是一个下标的列表，输出是对应的词嵌入。

参数：

- **num_embeddings** (*int*) - 嵌入字典的大小
- **embedding_dim** (*int*) - 每个嵌入向量的大小
- **padding_idx** (*int, optional*) - 如果提供的话，输出遇到此下标时用零填充
- **max_norm** (*float, optional*) - 如果提供的话，会重新归一化词嵌入，使它们的范数小于提供的值

- **norm_type** (*float*, *optional*) - 对于max_norm选项计算p范数时的p
- **scale_grad_by_freq** (*boolean*, *optional*) - 如果提供的话，会根据字典中单词频率缩放梯度

变量：

- **weight (*Tensor*)** -形状为(num_embeddings, embedding_dim)的模块中可学习的权值

形状：

- 输入： LongTensor *(N, W)*, N = mini-batch, W = 每个mini-batch中提取的下标数
- 输出： *(N, W, embedding_dim)*

例子：

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = Variable(torch.LongTensor([[1,2,4,5],[4,3,2,9]]))
>>> embedding(input)

Variable containing:
(0 ,.,.) =
 -1.0822  1.2522  0.2434
  0.8393 -0.6062 -0.3348
  0.6597  0.0350  0.0837
  0.5521  0.9447  0.0498

(1 ,.,.) =
  0.6597  0.0350  0.0837
 -0.1527  0.0877  0.4260
  0.8393 -0.6062 -0.3348
 -0.8738 -0.9054  0.4281
[torch.FloatTensor of size 2x4x3]

>>> # example with padding_idx
>>> embedding = nn.Embedding(10, 3, padding_idx=0)
>>> input = Variable(torch.LongTensor([[0,2,0,5]]))
>>> embedding(input)

Variable containing:
(0 ,.,.) =
  0.0000  0.0000  0.0000
  0.3452  0.4937 -0.9361
  0.0000  0.0000  0.0000
  0.0706 -2.1962 -0.6276
[torch.FloatTensor of size 1x4x3]
```

Distance functions

```
class torch.nn.PairwiseDistance(p=2, eps=1e-06)
```

按批计算向量v1, v2之间的距离：

$$\Vert x \Vert _p := \left(\sum_{i=1}^n \vert x_i \vert ^ p \right) ^ {1/p}$$

参数：

- **x** (*Tensor*): 包含两个输入batch的张量
- **p** (real): 范数次数，默认值：2

形状：

- 输入： \((N, D)\)，其中D=向量维数
- 输出： \((N, 1)\)

```
>>> pdist = nn.PairwiseDistance(2)
>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = pdist(input1, input2)
```

Loss functions

基本用法：

```
criterion = LossCriterion() #构造函数有自己的参数
loss = criterion(x, y) #调用标准时也有参数
```

计算出来的结果已经对 `mini-batch` 取了平均。

class torch.nn.L1Loss(size_average=True)[source]

创建一个衡量输入 `x` (`模型预测输出`)和目标 `y` 之间差的绝对值的平均值的标准。

$$ loss(x,y)=1/n\sum|x_i-y_i|

$$

- `x` 和 `y` 可以是任意形状，每个包含 `n` 个元素。
- 对 `n` 个元素对应的差值的绝对值求和，得出来的结果除以 `n` 。
- 如果在创建 `L1Loss` 实例的时候在构造函数中传入 `size_average=False` ，那么求出来的绝对值的和将不会除以 `n`

class torch.nn.MSELoss(size_average=True)[source]

创建一个衡量输入 `x` (`模型预测输出`)和目标 `y` 之间均方误差标准。

$$ loss(x,y)=1/n\sum(x_i-y_i)^2

$$

- `x` 和 `y` 可以是任意形状，每个包含 `n` 个元素。
- 对 `n` 个元素对应的差值的绝对值求和，得出来的结果除以 `n` 。
- 如果在创建 `MSELoss` 实例的时候在构造函数中传入 `size_average=False` ，那么求出来的平方和将不会除以 `n`

class torch.nn.CrossEntropyLoss(weight=None, size_average=True)[source]

此标准将 `LogSoftMax` 和 `NLLLoss` 集成到一个类中。

当训练一个多类分类器的时候，这个方法是十分有用的。

- weight(tensor): `1-D` tensor， `n` 个元素，分别代表 `n` 类的权重，如果你的训练样本很不均衡的话，是非常有用的。默认值为None。

调用时参数：

- input : 包含每个类的得分， `2-D` tensor, `shape` 为 `batch*n`
- target: 大小为 `n` 的 `1—D` `tensor` ，包含类别的索引(`0到 n-1`)。

Loss可以表述为以下形式：

$$ \begin{aligned} loss(x, class) &= -\text{log}\frac{exp(x[class])}{\sum_j exp(x[j]))}\ &= -x[class] + log(\sum_j exp(x[j])) \end{aligned}

$$ 当 `weight` 参数被指定的时候， `loss` 的计算公式变为：

$$ loss(x, class) = weights[class] * (-x[class] + log(\sum_j exp(x[j])))

$$ 计算出的 `loss` 对 `mini-batch` 的大小取了平均。

形状(`shape`)：

- Input: (N,C) `C` 是类别的数量
- Target: (N) `N` 是 `mini-batch` 的大小，0 <= targets[i] <= C-1

class torch.nn.NLLLoss(weight=None, size_average=True) [source]

负的 `log likelihood loss` 损失。用于训练一个 `n` 类分类器。

如果提供的话， `weight` 参数应该是一个 `1-D` tensor，里面的值对应类别的权重。当你的训练集样本不均衡的话，使用这个参数是非常有用的。

输入是一个包含类别 `log-probabilities` 的 `2-D` tensor，形状是 `（mini-batch， n）`

可以通过在最后一层加 `LogSoftmax` 来获得类别的 `log-probabilities` 。

如果您不想增加一个额外层的话，您可以使用 `CrossEntropyLoss` 。

此 `loss` 期望的 `target` 是类别的索引 (0 to N-1, where N = number of classes)

此 `loss` 可以被表示如下：

$$ loss(x, class) = -x[class]

$$ 如果 `weights` 参数被指定的话， `loss` 可以表示如下：

$$ loss(x, class) = -weights[class] * x[class]

$$ 参数说明：

- weight (Tensor, optional) – 手动指定每个类别的权重。如果给定的话，必须是长度为 `nclasses`
- size_average (bool, optional) – 默认情况下，会计算 `mini-batch``loss` 的平均值。然而，如果 `size_average=False` 那么将会把 `mini-batch` 中所有样本的 `loss` 累加起来。

形状:

- Input: (N,C) , `C` 是类别的个数
- Target: (N) ， `target` 中每个值的大小满足 `0 <= targets[i] <= C-1`

例子:

```
m = nn.LogSoftmax()
loss = nn.NLLLoss()
# input is of size nBatch x nClasses = 3 x 5
input = autograd.Variable(torch.randn(3, 5), requires_grad=True)
# each element in target has to have 0 <= value < nclasses
target = autograd.Variable(torch.LongTensor([1, 0, 4]))
output = loss(m(input), target)
output.backward()
```

class torch.nn.NLLLoss2d(weight=None, size_average=True)[source]

对于图片的 `negative log likehood loss` 。计算每个像素的 `NLL loss` 。

参数说明：

- weight (Tensor, optional) – 用来作为每类的权重，如果提供的话，必须为 `1-D` tensor，大小为 `C` ：类别的个数。
- size_average – 默认情况下，会计算 `mini-batch` loss均值。如果设置为 `False` 的话，将会累加 `mini-batch` 中所有样本的 `loss` 值。默认值： `True` 。

形状：

- Input: (N,C,H,W) `C` 类的数量
- Target: (N,H,W) where each value is 0 <= targets[i] <= C-1

例子：

```
m = nn.Conv2d(16, 32, (3, 3)).float()
loss = nn.NLLLoss2d()
# input is of size nBatch x nClasses x height x width
input = autograd.Variable(torch.randn(3, 16, 10, 10))
# each element in target has to have 0 <= value < nclasses
target = autograd.Variable(torch.LongTensor(3, 8, 8).random_(0, 4))
output = loss(m(input), target)
output.backward()
```

class torch.nn.KLDivLoss(weight=None, size_average=True)[source]

计算 KL 散度损失。

KL散度常用来描述两个分布的距离，并在输出分布的空间上执行直接回归是有用的。

与 `NLLLoss` 一样，给定的输入应该是 `log-probabilities` 。然而。和 `NLLLoss` 不同的是， `input` 不限于 `2-D` tensor，因为此标准是基于 `element` 的。

`target` 应该和 `input` 的形状相同。

此loss可以表示为：

$$ loss(x,target)=\frac{1}{n}\sum_i(target_i*(log(target_i)-x_i))

$$ 默认情况下，loss会基于 `element` 求平均。如果 `size_average=False` `loss` 会被累加起来。

class torch.nn.BCELoss(weight=None, size_average=True) [source]

计算 `target` 与 `output` 之间的二进制交叉熵。

$$ loss(o,t)=-\frac{1}{n}\sum_i(t[i] *log(o[i])+(1-t[i])* log(1-o[i]))

$$ 如果 `weight` 被指定 ：

$$ loss(o,t)=-\frac{1}{n}\sum_iweights[i] *(t[i]* log(o[i])+(1-t[i])* log(1-o[i]))

$$

这个用于计算 `auto-encoder` 的 `reconstruction error` 。注意 0<=target[i]<=1。

默认情况下，loss会基于`element`平均，如果`size_average=False`的话，`loss`会被累加。

class torch.nn.MarginRankingLoss(margin=0, size_average=True)[source]

创建一个标准，给定输入 $x1$,$x2$两个1-D mini-batch Tensor's，和一个y(1-D mini-batch tensor) ,y里面的值只能是-1或1。

如果 `y=1` ，代表第一个输入的值应该大于第二个输入的值，如果`y=-1`的话，则相反。

`mini-batch`中每个样本的loss的计算公式如下：

$$loss(x, y) = max(0, -y * (x1 - x2) + margin)$$

如果`size_average=True`,那么求出的`loss`将会对`mini-batch`求平均，反之，求出的`loss`会累加。默认情况下，`size_average=True`。

class torch.nn.HingeEmbeddingLoss(size_average=True) [source]

给定一个输入 x(2-D mini-batch tensor)和对应的 标签 y (1-D tensor,1,-1)，此函数用来计算之间的损失值。这个`loss`通常用来测量两个输入是否相似，即：使用L1 成对距离。典型是用在学习非线性`embedding`或者半监督学习中：

$$ loss(x,y)=\frac{1}{n}\sum_i \begin{cases} x_i, &\text if~y_i==1 \ max(0, margin-x_i), &if ~y_i==-1 \end{cases}

$$ x和y可以是任意形状，且都有`n`的元素，`loss`的求和操作作用在所有的元素上，然后除以`n`。如果您不想除以`n`的话，可以通过设置`size_average=False`。

`margin`的默认值为1,可以通过构造函数来设置。

class torch.nn.MultiLabelMarginLoss(size_average=True) [source]

计算多标签分类的`hinge loss`(`margin-based loss`)，计算`loss`时需要两个输入：input x(`2-D mini-batch Tensor`)，和 output y(`2-D tensor`表示mini-batch中样本类别的索引)。

$$ loss(x, y) = \frac{1}{x.size(0)}\sum_{i=0,j=0}^{I,J}(max(0, 1 - (x[y[j]] - x[i])))

$$ 其中`I=x.size(0),J=y.size(0)`。对于所有的`i`和`j`，满足 $y[j]\neq0, i \neq y[j]$

`x`和`y`必须具有同样的`size`。

这个标准仅考虑了第一个非零`y[j] targets`此标准允许了，对于每个样本来说，可以有多个类别。

class torch.nn.SmoothL1Loss(size_average=True)[source]

平滑版 `L1 loss` 。

loss的公式如下：

$$ loss(x, y) = \frac{1}{n}\sum_i \begin{cases} 0.5*(x_i-y_i)^2, & if~|x_i - y_i| < 1\ |x_i - y_i| - 0.5, & otherwise
\end{cases}

$$ 此loss对于异常点的敏感性不如 `MSELoss` ，而且，在某些情况下防止了梯度爆炸，(参照 `Fast R-CNN`)。这个 `loss` 有时也被称为 `Huber loss` 。

x 和 y 可以是任何包含 `n` 个元素的tensor。默认情况下，求出来的 `loss` 会除以 `n` ，可以通过设置 `size_average=True` 使loss累加。

class torch.nn.SoftMarginLoss(size_average=True)[source]

创建一个标准，用来优化2分类的 `logistic loss` 。输入为 `x` （一个 2-D mini-batch Tensor）和 目标 `y` （一个包含1或-1的Tensor）。

$$ loss(x, y) = \frac{1}{x.nelement()}\sum_i (log(1 + exp(-y[i]* x[i])))

$$ 如果求出的 `loss` 不想被平均可以通过设置 `size_average=False` 。

class torch.nn.MultiLabelSoftMarginLoss(weight=None, size_average=True)[source]

创建一个标准，基于输入x和目标y的 `max-entropy` ，优化多标签 `one-versus-all` 的损失。 `x` :2-D mini-batch Tensor; `y` :binary 2D Tensor。对每个mini-batch中的样本，对应的loss为：

$$ loss(x, y) = - \frac{1}{x.nElement()}\sum_{i=0}^I y[i]\text{log}\frac{exp(x[i])}{(1 + exp(x[i])}

```
                + (1-y[i])\text{log}\frac{1}{1+exp(x[i])}
```

$$ 其中 `I=x.nElement()-1` , $y[i] \in {0,1}$， `y` 和 `x` 必须要有同样 `size` 。

class torch.nn.CosineEmbeddingLoss(margin=0, size_average=True)[source]

给定 输入 `Tensors` ， `x1` , `x2` 和一个标签Tensor `y` (元素的值为1或-1)。此标准使用 `cosine` 距离测量两个输入是否相似，一般用来用来学习非线性 `embedding` 或者半监督学习。

`margin` 应该是-1到1之间的值，建议使用0到0.5。如果没有传入 `margin` 实参，默认值为0。

每个样本的loss是：

$$ loss(x, y) = \begin{cases} 1 - cos(x1, x2), &if~y == 1 \ max(0, cos(x1, x2) - margin), &if~y == -1 \end{cases}

$$ 如果 `size_average=True` 求出的loss会对batch求均值，如果 `size_average=False` 的话，则会累加 `loss` 。默认情况 `size_average=True` 。

class torch.nn.MultiMarginLoss(p=1, margin=1, weight=None, size_average=True)[source]

用来计算multi-class classification的hinge loss（magin-based loss）。输入是 `x` (2D mini-batch Tensor), `y` (1D Tensor)包含类别的索引， `0 <= y <= x.size(1))` 。

对每个mini-batch样本：

$$ loss(x, y) = \frac{1}{x.size(0)}\sum_{i=0}^I(max(0, margin - x[y] + x[i])^p)

$$ 其中 `I=x.size(0)` $i\neq y$。 可选择的，如果您不想所有的类拥有同样的权重的话，您可以通过在构造函数中传入 `weights` 参数来解决这个问题， `weights` 是一个1D权重Tensor。

传入weights后，loss函数变为：

$$ loss(x, y) = \frac{1}{x.size(0)}\sum_imax(0, w[y] * (margin - x[y] - x[i]))^p

$$ 默认情况下，求出的loss会对mini-batch取平均，可以通过设置 `size_average=False` 来取消取平均操作。

Vision layers

class torch.nn.PixelShuffle(upscale_factor)[source]

将shape为$[N, C*r^2, H, W]$的 `Tensor` 重新排列为*shape*为*$[N, C, H*r, W*r]$的Tensor。 当使用 `stride=1/r` 的sub-pixel卷积的时候，这个方法是非常有用的。

请看paperReal-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network by Shi et. al (2016) 获取详细信息。

参数说明：

- upscale_factor (int) – 增加空间分辨率的因子

Shape:

- Input: $[N,C*upscale_factor^2,H,W$]

- Output: $[N,C,H*upscale_factor,W*upscale_factor]$

例子:

```
>>> ps = nn.PixelShuffle(3)
>>> input = autograd.Variable(torch.Tensor(1, 9, 4, 4))
>>> output = ps(input)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

class torch.nn.UpsamplingNearest2d(size=None, scale_factor=None)[source]

对于多channel 输入 进行 `2-D` 最近邻上采样。

可以通过`size`或者`scale_factor`来指定上采样后的图片大小。

当给定`size`时，`size`的值将会是输出图片的大小。

参数：

- size (tuple, optional) – 一个包含两个整数的元组 (H_out, W_out)指定了输出的长宽
- scale_factor (int, optional) – 长和宽的一个乘子

形状：

- Input: (N,C,H_in,W_in)
- Output: (N,C,H_out,W_out) Hout=floor(H_in∗scale_factor) Wout=floor(W_in∗scale_factor)

例子：

```
>>> inp
Variable containing:
(0 ,0 ,.,.) =
  1  2
  3  4
[torch.FloatTensor of size 1x1x2x2]

>>> m = nn.UpsamplingNearest2d(scale_factor=2)
>>> m(inp)
Variable containing:
(0 ,0 ,.,.) =
  1  1  2  2
  1  1  2  2
  3  3  4  4
  3  3  4  4
[torch.FloatTensor of size 1x1x4x4]
```

class torch.nn.UpsamplingBilinear2d(size=None, scale_factor=None)[source]

对于多channel 输入 进行 `2-D` `bilinear` 上采样。

可以通过 `size` 或者 `scale_factor` 来指定上采样后的图片大小。

当给定 `size` 时， `size` 的值将会是输出图片的大小。

参数：

- size (tuple, optional) – 一个包含两个整数的元组 (H_out, W_out)指定了输出的长宽
- scale_factor (int, optional) – 长和宽的一个乘子

形状：

- Input: (N,C,H_in,W_in)
- Output: (N,C,H_out,W_out) Hout=floor(H_in∗scale_factor) Wout=floor(W_in∗scale_factor)

例子：

```
>>> inp
Variable containing:
(0 ,0 ,.,.) =
  1  2
  3  4
[torch.FloatTensor of size 1x1x2x2]

>>> m = nn.UpsamplingBilinear2d(scale_factor=2)
>>> m(inp)
Variable containing:
(0 ,0 ,.,.) =
  1.0000  1.3333  1.6667  2.0000
  1.6667  2.0000  2.3333  2.6667
  2.3333  2.6667  3.0000  3.3333
  3.0000  3.3333  3.6667  4.0000
[torch.FloatTensor of size 1x1x4x4]
```

Multi-GPU layers

class torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)[source]

在模块级别上实现数据并行。

此容器通过将 `mini-batch` 划分到不同的设备上来实现给定 `module` 的并行。在 `forward` 过程中， `module` 会在每个设备上都复制一遍，每个副本都会处理部分输入。在 `backward` 过程中，副本上的梯度会累加到原始 `module` 上。

batch的大小应该大于所使用的GPU的数量。还应当是GPU个数的整数倍，这样划分出来的每一块都会有相同的样本数量。

请看: Use nn.DataParallel instead of multiprocessing

除了 `Tensor` ，任何位置参数和关键字参数都可以传到DataParallel中。所有的变量会通过指定的 `dim` 来划分（默认值为0）。原始类型将会被广播，但是所有的其它类型都会被浅复制。所以如果在模型的 `forward` 过程中写入的话，将会被损坏。

参数说明：

- module – 要被并行的module
- device_ids – CUDA设备，默认为所有设备。
- output_device – 输出设备（默认为device_ids[0]）

例子：

```
net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
output = net(input_var)
```

Utilities

工具函数

torch.nn.utils.clip_grad_norm(parameters, max_norm, norm_type=2)[source]

Clips gradient norm of an iterable of parameters.

正则项的值由所有的梯度计算出来，就像他们连成一个向量一样。梯度被 `in-place operation` 修改。

参数说明:

- parameters (Iterable[Variable]) – 可迭代的 `Variables` ，它们的梯度即将被标准化。
- max_norm (float or int) – `clip` 后， `gradients` p-norm 值
- norm_type (float or int) – 标准化的类型，p-norm. 可以是 `inf` 代表 infinity norm.

关于norm

返回值:

所有参数的p-norm值。

torch.nn.utils.rnn.PackedSequence(_cls, data, batch_sizes)[source]

Holds the data and list of batch_sizes of a packed sequence.

All RNN modules accept packed sequences as inputs. 所有的 `RNN` 模块都接收这种被包裹后的序列作为它们的输入。

`NOTE：` 这个类的实例不能手动创建。它们只能被 `pack_padded_sequence()` 实例化。

参数说明:

- data (Variable) – 包含打包后序列的 `Variable` 。
- batch_sizes (list[int]) – 包含 `mini-batch` 中每个序列长度的列表。

torch.nn.utils.rnn.pack_padded_sequence(input, lengths, batch_first=False)[source]

这里的 `pack` ，理解成压紧比较好。 将一个 填充过的变长序列 压紧。（填充时候，会有冗余，所以压紧一下）

输入的形状可以是(T×B×)。 `T` 是最长序列长度， `B` 是 `batch size` ，` `代表任意维度(可以是0)。如果` batch_first=True `的话，那么相应的` input size `就是` (B×T×*)`。

`Variable` 中保存的序列，应该按序列长度的长短排序，长的在前，短的在后。即 `input[:,0]` 代表的是最长的序列， `input[:, B-1]` 保存的是最短的序列。

`NOTE：` 只要是维度大于等于2的 `input` 都可以作为这个函数的参数。你可以用它来打包 `labels` ，然后用 `RNN` 的输出和打包后的 `labels` 来计算 `loss` 。通过 `PackedSequence` 对象的 `.data` 属性可以获取 `Variable` 。

参数说明:

- input (Variable) – 变长序列 被填充后的 batch
- lengths (list[int]) – `Variable` 中 每个序列的长度。
- batch_first (bool, optional) – 如果是 `True` ，input的形状应该是 `B*T*size` 。

返回值:

一个 `PackedSequence` 对象。

torch.nn.utils.rnn.pad_packed_sequence(sequence, batch_first=False)[source]

填充 `packed_sequence` 。

上面提到的函数的功能是将一个填充后的变长序列压紧。 这个操作和pack_padded_sequence()是相反的。把压紧的序列再填充回来。

返回的Varaible的值的 `size` 是 `T×B×*` ， `T` 是最长序列的长度， `B` 是 batch_size,如果 `batch_first=True` ,那么返回值是 `B×T×*` 。

Batch中的元素将会以它们长度的逆序排列。

参数说明:

- sequence (PackedSequence) – 将要被填充的 batch
- batch_first (bool, optional) – 如果为True，返回的数据的格式为 `B×T×*` 。

返回值: 一个tuple，包含被填充后的序列，和batch中序列的长度列表。

例子：

```
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.nn import utils as nn_utils
batch_size = 2
max_length = 3
hidden_size = 2
n_layers =1

tensor_in = torch.FloatTensor([[1, 2, 3], [1, 0, 0]]).resize_(2,3,1)
tensor_in = Variable( tensor_in ) #[batch, seq, feature], [2, 3, 1]
seq_lengths = [3,1] # list of integers holding information about the batch size at eac
h sequence step

# pack it
pack = nn_utils.rnn.pack_padded_sequence(tensor_in, seq_lengths, batch_first=True)

# initialize
rnn = nn.RNN(1, hidden_size, n_layers, batch_first=True)
h0 = Variable(torch.randn(n_layers, batch_size, hidden_size))

#forward
out, _ = rnn(pack, h0)

# unpack
unpacked = nn_utils.rnn.pad_packed_sequence(out)
print(unpacked)
```

关于packed_sequence

torch.nn.functional

Convolution 函数

```
torch.nn.functional.conv1d(input, weight, bias=None, stride=1, padding=0, dilation=1,
groups=1)
```

对几个输入平面组成的输入信号应用1D卷积。

有关详细信息和输出形状，请参见 `Conv1d` 。

参数：

- **input** – 输入张量的形状 (minibatch x in_channels x iW)
- **weight** – 过滤器的形状 (out_channels, in_channels, kW)
- **bias** – 可选偏置的形状 (out_channels)
- **stride** – 卷积核的步长，默认为1

例子：

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3))
>>> inputs = autograd.Variable(torch.randn(20, 16, 50))
>>> F.conv1d(inputs, filters)
```

```
torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1,
groups=1)
```

对几个输入平面组成的输入信号应用2D卷积。

有关详细信息和输出形状，请参见 `Conv2d` 。

参数：

- **input** – 输入张量 (minibatch x in_channels x iH x iW)
- **weight** – 过滤器张量 (out_channels, in_channels/groups, kH, kW)
- **bias** – 可选偏置张量 (out_channels)
- **stride** – 卷积核的步长，可以是单个数字或一个元组 (sh x sw)。默认为1
- **padding** – 输入上隐含零填充。可以是单个数字或元组。 默认值：0
- **groups** – 将输入分成组，in_channels应该被组数除尽

例子：

```
>>> # With square kernels and equal stride
>>> filters = autograd.Variable(torch.randn(8,4,3,3))
>>> inputs = autograd.Variable(torch.randn(1,4,5,5))
>>> F.conv2d(inputs, filters, padding=1)
```

```
torch.nn.functional.conv3d(input, weight, bias=None, stride=1, padding=0, dilation=1,
groups=1)
```

对几个输入平面组成的输入信号应用3D卷积。

有关详细信息和输出形状，请参见 `Conv3d` 。

参数：

- **input** – 输入张量的形状 (minibatch x in_channels x iT x iH x iW)
- **weight** – 过滤器张量的形状 (out_channels, in_channels, kT, kH, kW)
- **bias** – 可选偏置张量的形状 (out_channels)
- **stride** – 卷积核的步长，可以是单个数字或一个元组 (sh x sw)。默认为1
- **padding** – 输入上隐含零填充。可以是单个数字或元组。 默认值：0

例子：

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3, 3, 3))
>>> inputs = autograd.Variable(torch.randn(20, 16, 50, 10, 20))
>>> F.conv3d(inputs, filters)
```

```
torch.nn.functional.conv_transpose1d(input, weight, bias=None, stride=1, padding=0, ou
tput_padding=0, groups=1)
```

```
torch.nn.functional.conv_transpose2d(input, weight, bias=None, stride=1, padding=0, ou
tput_padding=0, groups=1)
```

在由几个输入平面组成的输入图像上应用二维转置卷积，有时也称为“去卷积”。

有关详细信息和输出形状，请参阅 `ConvTranspose2d` 。

参数：

- **input** – 输入张量的形状 (minibatch x in_channels x iH x iW)
- **weight** – 过滤器的形状 (in_channels x out_channels x kH x kW)
- **bias** – 可选偏置的形状 (out_channels)
- **stride** – 卷积核的步长，可以是单个数字或一个元组 (sh x sw)。默认: 1
- **padding** – 输入上隐含零填充。可以是单个数字或元组。 (padh x padw)。默认: 0
- **groups** – 将输入分成组， `in_channels` 应该被组数除尽
- **output_padding** – 0 <= padding <stride的零填充，应该添加到输出。可以是单个数字或

元组。默认值：0

```
torch.nn.functional.conv_transpose3d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1)
```

在由几个输入平面组成的输入图像上应用三维转置卷积，有时也称为“去卷积”。

有关详细信息和输出形状，请参阅 `ConvTranspose3d` 。

参数：

- **input** – 输入张量的形状 (minibatch x in_channels x iT x iH x iW)
- **weight** – 过滤器的形状 (in_channels x out_channels x kH x kW)
- **bias** – 可选偏置的形状 (out_channels)
- **stride** – 卷积核的步长，可以是单个数字或一个元组 (sh x sw)。默认: 1
- **padding** – 输入上隐含零填充。可以是单个数字或元组。 (padh x padw)。默认: 0

Pooling 函数

```
torch.nn.functional.avg_pool1d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)
```

对由几个输入平面组成的输入信号进行一维平均池化。

有关详细信息和输出形状，请参阅 `AvgPool1d` 。

参数：

- **kernel_size** – 窗口的大小
- **stride** – 窗口的步长。默认值为 `kernel_size`
- **padding** – 在两边添加隐式零填充
- **ceil_mode** – 当为True时，将使用 `ceil` 代替 `floor` 来计算输出形状
- **count_include_pad** – 当为True时，这将在平均计算时包括补零

例子：

```
>>> # pool of square window of size=3, stride=2
>>> input = Variable(torch.Tensor([[[1,2,3,4,5,6,7]]]))
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
Variable containing:
(0 ,.,.) =
  2  4  6
[torch.FloatTensor of size 1x1x3]
```

```
torch.nn.functional.avg_pool2d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)
```

在kh x kw区域中应用步长为dh x dw的二维平均池化操作。输出特征的数量等于输入平面的数量。

有关详细信息和输出形状，请参阅 `AvgPool2d` 。

参数：

- **input** – 输入的张量 (minibatch x in_channels x iH x iW)
- **kernel_size** – 池化区域的大小，可以是单个数字或者元组 (kh x kw)
- **stride** – 池化操作的步长，可以是单个数字或者元组 (sh x sw)。默认等于核的大小
- **padding** – 在输入上隐式的零填充，可以是单个数字或者一个元组 (padh x padw)，默认: 0
- **ceil_mode** – 定义空间输出形状的操作
- **count_include_pad** – 除以原始非填充图像内的元素数量或kh * kw

```
torch.nn.functional.avg_pool3d(input, kernel_size, stride=None)
```

在kt x kh x kw区域中应用步长为dt x dh x dw的二维平均池化操作。输出特征的数量等于 input planes / dt。

```
torch.nn.functional.max_pool1d(input, kernel_size, stride=None, padding=0, dilation=1,
 ceil_mode=False, return_indices=False)
```

```
torch.nn.functional.max_pool2d(input, kernel_size, stride=None, padding=0, dilation=1,
 ceil_mode=False, return_indices=False)
```

```
torch.nn.functional.max_pool3d(input, kernel_size, stride=None, padding=0, dilation=1,
 ceil_mode=False, return_indices=False)
```

```
torch.nn.functional.max_unpool1d(input, indices, kernel_size, stride=None, padding=0,
output_size=None)
```

```
torch.nn.functional.max_unpool2d(input, indices, kernel_size, stride=None, padding=0,
output_size=None)
```

```
torch.nn.functional.max_unpool3d(input, indices, kernel_size, stride=None, padding=0,
output_size=None)
```

```
torch.nn.functional.lp_pool2d(input, norm_type, kernel_size, stride=None, ceil_mode=Fa
lse)
```

```
torch.nn.functional.adaptive_max_pool1d(input, output_size, return_indices=False)
```

在由几个输入平面组成的输入信号上应用1D自适应最大池化。

有关详细信息和输出形状，请参阅 `AdaptiveMaxPool1d` 。

参数：

- **output_size** – 目标输出大小（单个整数）
- **return_indices** – 是否返回池化的指数

```
torch.nn.functional.adaptive_max_pool2d(input, output_size, return_indices=False)
```

在由几个输入平面组成的输入信号上应用2D自适应最大池化。

有关详细信息和输出形状，请参阅 `AdaptiveMaxPool2d` 。

参数：

- **output_size** – 目标输出大小（单整数或双整数元组）
- **return_indices** – 是否返回池化的指数

```
torch.nn.functional.adaptive_avg_pool1d(input, output_size)
```

在由几个输入平面组成的输入信号上应用1D自适应平均池化。

有关详细信息和输出形状，请参阅 `AdaptiveAvgPool1d` 。

参数：

- **output_size** – 目标输出大小（单整数或双整数元组）

```
torch.nn.functional.adaptive_avg_pool2d(input, output_size)
```

在由几个输入平面组成的输入信号上应用2D自适应平均池化。

有关详细信息和输出形状，请参阅 `AdaptiveAvgPool2d` 。

参数：

- **output_size** – 目标输出大小（单整数或双整数元组）

非线性激活函数

```
torch.nn.functional.threshold(input, threshold, value, inplace=False)
```

```
torch.nn.functional.relu(input, inplace=False)
```

```
torch.nn.functional.hardtanh(input, min_val=-1.0, max_val=1.0, inplace=False)
```

```
torch.nn.functional.relu6(input, inplace=False)
```

```
torch.nn.functional.elu(input, alpha=1.0, inplace=False)
```

```
torch.nn.functional.leaky_relu(input, negative_slope=0.01, inplace=False)
```

```
torch.nn.functional.prelu(input, weight)
```

```
torch.nn.functional.rrelu(input, lower=0.125, upper=0.3333333333333333, training=False
, inplace=False)
```

```
torch.nn.functional.logsigmoid(input)
```

```
torch.nn.functional.hardshrink(input, lambd=0.5)
```

```
torch.nn.functional.tanhshrink(input)
```

```
torch.nn.functional.softsign(input)
```

```
torch.nn.functional.softplus(input, beta=1, threshold=20)
```

```
torch.nn.functional.softmin(input)
```

```
torch.nn.functional.softmax(input)
```

```
torch.nn.functional.softshrink(input, lambd=0.5)
```

```
torch.nn.functional.log_softmax(input)
```

```
torch.nn.functional.tanh(input)
```

```
torch.nn.functional.sigmoid(input)
```

Normalization 函数

```
torch.nn.functional.batch_norm(input, running_mean, running_var, weight=None, bias=None
, training=False, momentum=0.1, eps=1e-05)
```

线性函数

```
torch.nn.functional.linear(input, weight, bias=None)
```

Dropout 函数

```
torch.nn.functional.dropout(input, p=0.5, training=False, inplace=False)
```

距离函数（Distance functions）

```
torch.nn.functional.pairwise_distance(x1, x2, p=2, eps=1e-06)
```

计算向量v1、v2之间的距离（成次或者成对，意思是可以计算多个，可以参看后面的参数）

$$\left | x \right |_{p}:=\left (\sum_{i=1}^{N}\left | x_{i}^{p} \right | \right)^{1/p}$$

参数：

- x1:第一个输入的张量
- x2:第二个输入的张量
- p:矩阵范数的维度。默认值是2，即二范数。

规格：

- 输入:(N,D)其中D等于向量的维度
- 输出:(N,1)

例子：

```
>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = F.pairwise_distance(input1, input2, p=2)
>>> output.backward()
```

损失函数（Loss functions）

```
torch.nn.functional.nll_loss(input, target, weight=None, size_average=True)
```

负的log likelihood损失函数. 详细请看NLLLoss.

参数：

- **input** - (N,C) C 是类别的个数
- **target** - (N) 其大小是 0 <= targets[i] <= C-1
- **weight** (Variable, optional) – 一个可手动指定每个类别的权重。如果给定的话，必须是大小为nclasses的Variable
- **size_average** (bool, optional) – 默认情况下，是`mini-batch` loss的平均值，然而，如果size_average=False，则是`mini-batch` loss的总和。

Variables:

- **weight** – 对于constructor而言，每一类的权重作为输入

```
torch.nn.functional.kl_div(input, target, size_average=True)
```

KL 散度损失函数，详细请看KLDivLoss

参数：

- **input** – 任意形状的 Variable
- **target** – 与输入相同形状的 Variable
- **size_average** – 如果为TRUE，loss则是平均值，需要除以输入 tensor 中 element 的数目

```
torch.nn.functional.cross_entropy(input, target, weight=None, size_average=True)
```

该函数使用了 log_softmax 和 nll_loss，详细请看CrossEntropyLoss

参数：

- **input** - (N,C) 其中，C 是类别的个数
- **target** - (N) 其大小是 0 <= targets[i] <= C-1
- **weight** (Variable, optional) – 一个可手动指定每个类别的权重。如果给定的话，必须是大

小为nclasses的Variable

- **size_average** (bool, optional) – 默认情况下，是`mini-batch`loss的平均值，然而，如果size_average=False，则是`mini-batch`loss的总和。

```
torch.nn.functional.binary_cross_entropy(input, target, weight=None, size_average=True
)
```

该函数计算了输出与target之间的二进制交叉熵，详细请看BCELoss

参数：

- **input** – 任意形状的 Variable
- **target** – 与输入相同形状的 Variable
- **weight** (Variable, optional) – 一个可手动指定每个类别的权重。如果给定的话，必须是大小为nclasses的Variable
- **size_average** (bool, optional) – 默认情况下，是`mini-batch`loss的平均值，然而，如果size_average=False，则是`mini-batch`loss的总和。

```
torch.nn.functional.smooth_l1_loss(input, target, size_average=True)
```

Vision functions

torch.nn.functional.pixel_shuffle(input, upscale_factor)[source]

将形状为`[*, C*r^2, H, W]`的`Tensor`重新排列成形状为`[C, H*r, W*r]`的Tensor.

详细请看PixelShuffle.

形参说明:

- input (Variable) – 输入
- upscale_factor (int) – 增加空间分辨率的因子.

例子:

```
ps = nn.PixelShuffle(3)
input = autograd.Variable(torch.Tensor(1, 9, 4, 4))
output = ps(input)
print(output.size())
torch.Size([1, 1, 12, 12])
```

torch.nn.functional.pad(input, pad, mode='constant', value=0)[source]

填充 `Tensor` .

目前为止,只支持 `2D` 和 `3D` 填充. Currently only 2D and 3D padding supported. 当输入为 `4D Tensor` 的时候, `pad` 应该是一个4元素的 `tuple (pad_l, pad_r, pad_t, pad_b )` ,当输入为 `5D Tensor` 的时候, `pad` 应该是一个6元素的 `tuple (pleft, pright, ptop, pbottom, pfront, pback)` .

形参说明:

- input (Variable) – 4D 或 5D `tensor`
- pad (tuple) – 4元素 或 6-元素 `tuple`
- mode – 'constant', 'reflect' or 'replicate'
- value – 用于 `constant padding` 的值.

Automatic differentiation package - torch.autograd

`torch.autograd` 提供了类和函数用来对任意标量函数进行求导。要想使用自动求导，只需要对已有的代码进行微小的改变。只需要将所有的 `tensor` 包含进 `Variable` 对象中即可。

torch.autograd.backward(variables, grad_variables, retain_variables=False)

Computes the sum of gradients of given variables w.r.t. graph leaves. 给定图的叶子节点 `variables`, 计算图中变量的梯度和。 计算图可以通过链式法则求导。如果 `variables` 中的任何一个 `variable` 是 非标量(`non-scalar`)的，且 `requires_grad=True` 。那么此函数需要指定 `grad_variables` ，它的长度应该和 `variables` 的长度匹配，里面保存了相关 `variable` 的梯度(对于不需要 `gradient tensor` 的 `variable` ， `None` 是可取的)。

此函数累积 `leaf variables` 计算的梯度。你可能需要在调用此函数之前将 `leaf variable` 的梯度置零。

参数说明:

- variables (variable 列表) – 被求微分的叶子节点，即 `ys` 。
- grad_variables (`Tensor` 列表) – 对应 `variable` 的梯度。仅当 `variable` 不是标量且需要求梯度的时候使用。
- retain_variables (bool) – `True` ,计算梯度时所需要的 `buffer` 在计算完梯度后不会被释放。如果想对一个子图多次求微分的话，需要设置为 `True` 。

Variable

API 兼容性

`Variable API` 几乎和 `Tensor API` 一致 (除了一些 `in-place` 方法，这些 `in-place` 方法会修改 `required_grad=True` 的 `input` 的值)。多数情况下，将 `Tensor` 替换为 `Variable` ，代码一样会正常的工作。由于这个原因，我们不会列出 `Variable` 的所有方法，你可以通过 `torch.Tensor` 的文档来获取相关知识。

In-place operations on Variables

在 `autograd` 中支持 `in-place operations` 是非常困难的。同时在很多情况下，我们阻止使用 `in-place operations` 。 `Autograd` 的贪婪的 释放 `buffer` 和 复用使得它效率非常高。只有在非常少的情况下，使用 `in-place operations` 可以降低内存的使用。除非你面临很大的内存压力，否则不要使用 `in-place operations` 。

In-place 正确性检查

所有的 `Variable` 都会记录用在他们身上的 `in-place operations` 。如果 `pytorch` 检测到 `variable` 在一个 `Function` 中已经被保存用来 `backward` ，但是之后它又被 `in-place operations` 修改。当这种情况发生时，在 `backward` 的时候， `pytorch` 就会报错。这种机制保证了，如果你用了 `in-place operations` ，但是在 `backward` 过程中没有报错，那么梯度的计算就是正确的。

class torch.autograd.Variable [source]

包装一个 `Tensor` ,并记录用在它身上的 `operations` 。

`Variable` 是 `Tensor` 对象的一个 `thin wrapper` ，它同时保存着 `Variable` 的梯度和创建这个 `Variable` 的 `Function` 的引用。这个引用可以用来追溯创建这个 `Variable` 的整条链。如果 `Variable` 是被用户所创建的，那么它的 `creator` 是 `None` ，我们称这种对象为 `leaf Variables` 。

由于 `autograd` 只支持标量值的反向求导(即： `y` 是标量)，梯度的大小总是和数据的大小匹配。同时，仅仅给 `leaf variables` 分配梯度，其他 `Variable` 的梯度总是为0.

`变量：`

- data – 包含的 `Tensor`
- grad – 保存着 `Variable` 的梯度。这个属性是懒分配的，且不能被重新分配。
- requires_grad – 布尔值，指示这个 `Variable` 是否是被一个包含 `Variable` 的子图创建的。更多细节请看 `Excluding subgraphs from backward` 。只能改变 `leaf variable` 的这个标签。
- volatile – 布尔值，指示这个 `Variable` 是否被用于推断模式(即，不保存历史信息)。更多细节请看 `Excluding subgraphs from backward` 。只能改变 `leaf variable` 的这个标签。
- creator – 创建这个 `Variable` 的 `Function` ，对于 `leaf variable` ，这个属性为 `None` 。只读属性。

`属性:`

- data (any tensor class) – 被包含的 `Tensor`
- requires_grad (bool) – `requires_grad` 标记. 只能通过 `keyword` 传入.

- volatile (bool) – `volatile` 标记. 只能通过 `keyword` 传入.

backward(gradient=None, retain_variables=False)[source]

当前 `Variable` 对 `leaf variable` 求偏导。

计算图可以通过链式法则求导。如果 `Variable` 是 非标量(`non-scalar`)的，且 `requires_grad=True` 。那么此函数需要指定 `gradient` ，它的形状应该和 `Variable` 的长度匹配，里面保存了 `Variable` 的梯度。

此函数累积 `leaf variable` 的梯度。你可能需要在调用此函数之前将 `Variable` 的梯度置零。

`参数:`

- gradient (Tensor) – 其他函数对于此 `Variable` 的导数。仅当 `Variable` 不是标量的时候使用，类型和位形状应该和 `self.data` 一致。
- retain_variables (bool) – `True` , 计算梯度所必要的 `buffer` 在经历过一次 `backward` 过程后不会被释放。如果你想多次计算某个子图的梯度的时候，设置为 `True` 。在某些情况下，使用 `autograd.backward()` 效率更高。

detach()[source]

Returns a new Variable, detached from the current graph. 返回一个新的 `Variable` ，从当前图中分离下来的。

返回的 `Variable` `requires_grad=False` ，如果输入 `volatile=True` ，那么返回的 `Variable` `volatile=True` 。

`注意：`

返回的 `Variable` 和原始的 `Variable` 公用同一个 `data tensor` 。 `in-place` 修改会在两个 `Variable` 上同时体现(因为它们共享 `data tensor`)，可能会导致错误。

detach_()[source]

将一个 `Variable` 从创建它的图中分离，并把它设置成 `leaf variable` 。

register_hook(hook)[source]

注册一个 `backward` 钩子。

每次 `gradients` 被计算的时候，这个 `hook` 都被调用。 `hook` 应该拥有以下签名：

`hook(grad) -> Variable or None`

`hook` 不应该修改它的输入，但是它可以选择性的返回一个替代当前梯度的新梯度。

这个函数返回一个 句柄(`handle`)。它有一个方法 `handle.remove()` ，可以用这个方法将 `hook` 从 `module` 移除。

Example

```
v = Variable(torch.Tensor([0, 0, 0]), requires_grad=True)
h = v.register_hook(lambda grad: grad * 2)  # double the gradient
v.backward(torch.Tensor([1, 1, 1]))
#先计算原始梯度，再进hook，获得一个新梯度。
print(v.grad.data)

 2
 2
 2
[torch.FloatTensor of size 3]
>>> h.remove()  # removes the hook
```

```
def w_hook(grad):
    print("hello")
    return None
w1 = Variable(torch.FloatTensor([1, 1, 1]),requires_grad=True)

w1.register_hook(w_hook) # 如果hook返回的是None的话，那么梯度还是原来计算的梯度。

w1.backward(gradient=torch.FloatTensor([1, 1, 1]))
print(w1.grad)
```

```
hello
Variable containing:
 1
 1
 1
[torch.FloatTensor of size 3]
```

reinforce(reward)[source]

注册一个奖励，这个奖励是由一个随机过程得到的。

微分一个随机节点需要提供一个奖励值。如果你的计算图中包含随机 `operations` ，你需要在他们的输出上调用这个函数。否则的话，会报错。

`参数:`

- reward (Tensor) – 每个元素的reward。必须和 `Varaible` 形状相同，并在同一个设备上。

class torch.autograd.Function[source]

Records operation history and defines formulas for differentiating ops. 记录 `operation` 的历史，定义微分公式。 每个执行在 `Varaibles` 上的 `operation` 都会创建一个 `Function` 对象，这个 `Function` 对象执行计算工作，同时记录下来。这个历史以有向无环图的形式保存下来，有向图的节点为 `functions` ，有向图的边代表数据依赖关系(`input<-output`)。之后，当 `backward` 被调用的时候，计算图以拓扑顺序处理，通过调用每个 `Function` 对象的 `backward()` ，同时将返回的梯度传递给下一个 `Function` 。

通常情况下，用户能和 `Functions` 交互的唯一方法就是创建 `Function` 的子类，定义新的 `operation`。这是扩展 `torch.autograd` 的推荐方法。

由于 `Function` 逻辑在很多脚本上都是热点，所有我们把几乎所有的 `Function` 都使用 `C` 实现，通过这种策略保证框架的开销是最小的。

每个 `Function` 只被使用一次(在forward过程中)。

`变量:`

- saved_tensors – 调用 `forward()` 时需要被保存的 `Tensors` 的 `tuple`。
- needs_input_grad – 长度为 输入数量的 布尔值组成的 `tuple`。指示给定的 `input` 是否需要梯度。这个被用来优化用于 `backward` 过程中的 `buffer`，忽略 `backward` 中的梯度计算。
- num_inputs – `forward` 的输入参数数量。
- num_outputs – `forward` 返回的 `Tensor` 数量。
- requires_grad – 布尔值。指示 `backward` 以后会不会被调用。
- previous_functions – 长度为 `num_inputs` 的 Tuple of (int, Function) pairs。`Tuple` 中的每单元保存着创建 `input` 的 `Function` 的引用，和索引。

backward(* grad_output)[source]

定义了 `operation` 的微分公式。

所有的 `Function` 子类都应该重写这个方法。

所有的参数都是 `Tensor`。他必须接收和 `forward` 的输出 相同个数的参数。而且它需要返回和 `forward` 的输入参数相同个数的 `Tensor`。 即： `backward` 的输入参数是 此 `operation` 的输出的值的梯度。`backward` 的返回值是此 `operation` 输入值的梯度。

forward(* input)[source]

执行 `operation`。

所有的 `Function` 子类都需要重写这个方法。

可以接收和返回任意个数 `tensors`

mark_dirty(* args)[source]

将输入的 `tensors` 标记为被 `in-place operation` 修改过。

这个方法应当至多调用一次，仅仅用在 `forward` 方法里，而且 `mark_dirty` 的实参只能是 `forward` 的实参。

每个在`forward`方法中被`in-place operations`修改的`tensor`都应该传递给这个方法。这样，可以保证检查的正确性。这个方法在`tensor`修改前后调用都可以。

mark_non_differentiable(* args)[source]

将输出标记为不可微。

这个方法至多只能被调用一次，只能在`forward`中调用，而且实参只能是`forward`的返回值。

这个方法会将输出标记成不可微，会增加`backward`过程中的效率。在`backward`中，你依旧需要接收`forward`输出值的梯度，但是这些梯度一直是`None`。

This is used e.g. for indices returned from a max Function.

mark_shared_storage(* pairs)[source]

将给定的`tensors pairs`标记为共享存储空间。

这个方法至多只能被调用一次，只能在`forward`中调用，而且所有的实参必须是`(input, output)`对。

如果一些 `inputs` 和 `outputs` 是共享存储空间的，所有的这样的`(input, output)`对都应该传给这个函数，保证 `in-place operations` 检查的正确性。唯一的特例就是，当`output`和`input`是同一个`tensor`(`in-place operations`的输入和输出)。这种情况下，就没必要指定它们之间的依赖关系，因为这个很容易就能推断出来。

这个函数在很多时候都用不到。主要是用在 索引 和 转置 这类的 `op` 中。

save_for_backward(* tensors)[source]

将传入的 `tensor` 保存起来，留着`backward`的时候用。

这个方法至多只能被调用一次，只能在`forward`中调用。

之后，被保存的`tensors`可以通过 `saved_tensors`属性获取。在返回这些`tensors`之前，`pytorch`做了一些检查，保证这些`tensor`没有被`in-place operations`修改过。

实参可以是`None`。

torch.optim

`torch.optim` 是一个实现了各种优化算法的库。大部分常用的方法得到支持，并且接口具备足够的通用性，使得未来能够集成更加复杂的方法。

如何使用optimizer

为了使用 `torch.optim` ，你需要构建一个optimizer对象。这个对象能够保持当前参数状态并基于计算得到的梯度进行参数更新。

构建

为了构建一个 `Optimizer` ，你需要给它一个包含了需要优化的参数（必须都是 `Variable` 对象）的iterable。然后，你可以设置optimizer的参 数选项，比如学习率，权重衰减，等等。

例子：

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr = 0.0001)
```

为每个参数单独设置选项

`Optimizer` 也支持为每个参数单独设置选项。若想这么做，不要直接传入 `Variable` 的iterable，而是传入 `dict` 的iterable。每一个dict都分别定 义了一组参数，并且包含一个 `param` 键，这个键对应参数的列表。其他的键应该optimizer所接受的其他参数的关键字相匹配，并且会被用于对这组参数的 优化。

`注意：`

你仍然能够传递选项作为关键字参数。在未重写这些选项的组中，它们会被用作默认值。当你只想改动一个参数组的选项，但其他参数组的选项不变时，这是 非常有用的。

例如，当我们想指定每一层的学习率时，这是非常有用的：

```
optim.SGD([
                {'params': model.base.parameters()},
                {'params': model.classifier.parameters(), 'lr': 1e-3}
            ], lr=1e-2, momentum=0.9)
```

这意味着 `model.base` 的参数将会使用 `1e-2` 的学习率， `model.classifier` 的参数将会使用 `1e-3` 的学习率，并且 `0.9` 的momentum将会被用于所 有的参数。

进行单次优化

所有的optimizer都实现了 `step()` 方法，这个方法会更新所有的参数。它能按两种方式来使用：

`optimizer.step()`

这是大多数optimizer所支持的简化版本。一旦梯度被如 `backward()` 之类的函数计算好后，我们就可以调用这个函数。

例子

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

`optimizer.step(closure)`

一些优化算法例如Conjugate Gradient和LBFGS需要重复多次计算函数，因此你需要传入一个闭包去允许它们重新计算你的模型。这个闭包应当清空梯度， 计算损失，然后返回。

例子：

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
    optimizer.step(closure)
```

算法

class torch.optim.Optimizer(params, defaults) [source]

Base class for all optimizers.

参数：

- params (iterable) —— `Variable` 或者 `dict` 的iterable。指定了什么参数应当被优化。
- defaults —— (dict)：包含了优化选项默认值的字典（一个参数组没有指定的参数选项将会使用默认值）。

load_state_dict(state_dict) [source]

加载optimizer状态

参数：

state_dict (`dict`) —— optimizer的状态。应当是一个调用 `state_dict()` 所返回的对象。

state_dict() [source]

以 `dict` 返回optimizer的状态。

它包含两项。

- state - 一个保存了当前优化状态的dict。optimizer的类别不同，state的内容也会不同。
- param_groups - 一个包含了全部参数组的dict。

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

zero_grad() [source]

清空所有被优化过的Variable的梯度.

class torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)[source]

实现Adadelta算法。

它在ADADELTA: An Adaptive Learning Rate Method.中被提出。

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- rho (`float` , 可选) – 用于计算平方梯度的运行平均值的系数（默认：0.9）
- eps (`float` , 可选) – 为了增加数值计算的稳定性而加到分母里的项（默认：1e-6）
- lr (`float` , 可选) – 在delta被应用到参数更新之前对它缩放的系数（默认：1.0）
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认: 0）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选

的。

class torch.optim.Adagrad(params, lr=0.01, lr_decay=0, weight_decay=0)[source]

实现Adagrad算法。

它在 Adaptive Subgradient Methods for Online Learning and Stochastic Optimization中被提出。

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float` , 可选) – 学习率（默认: 1e-2）
- lr_decay (`float` , 可选) – 学习率衰减（默认: 0）
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认: 0）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)[source]

实现Adam算法。

它在Adam: A Method for Stochastic Optimization中被提出。

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float` , 可选) – 学习率（默认：1e-3）
- betas (Tuple[`float` , `float`], 可选) – 用于计算梯度以及梯度平方的运行平均值的系数（默认：0.9，0.999）
- eps (`float` , 可选) – 为了增加数值计算的稳定性而加到分母里的项（默认：1e-8）
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认: 0）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.Adamax(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)[source]

实现Adamax算法（Adam的一种基于无穷范数的变种）。

它在Adam: A Method for Stochastic Optimization中被提出。

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float` , 可选) – 学习率（默认：2e-3）
- betas (Tuple[`float` , `float`], 可选) – 用于计算梯度以及梯度平方的运行平均值的系数
- eps (`float` , 可选) – 为了增加数值计算的稳定性而加到分母里的项（默认：1e-8）
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认: 0）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.ASGD(params, lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight_decay=0)[source]

实现平均随机梯度下降算法。

它在Acceleration of stochastic approximation by averaging中被提出。

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float` , 可选) – 学习率（默认：1e-2）
- lambd (`float` , 可选) – 衰减项（默认：1e-4）
- alpha (`float` , 可选) – eta更新的指数（默认：0.75）
- t0 (`float` , 可选) – 指明在哪一次开始平均化（默认：1e6）
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认: 0）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.LBFGS(params, lr=1, max_iter=20, max_eval=None, tolerance_grad=1e-05, tolerance_change=1e-09, history_size=100, line_search_fn=None)[source]

实现L-BFGS算法。

警告

这个optimizer不支持为每个参数单独设置选项以及不支持参数组（只能有一个）

警告

目前所有的参数不得不都在同一设备上。在将来这会得到改进。

注意

这是一个内存高度密集的optimizer（它要求额外的 `param_bytes * (history_size + 1)` 个字节）。如果它不适应内存，尝试减小history size，或者使用不同的算法。

参数：

- lr (`float`) – 学习率（默认：1）
- max_iter (`int`) – 每一步优化的最大迭代次数（默认：20））
- max_eval (`int`) – 每一步优化的最大函数评价次数（默认：max * 1.25）
- tolerance_grad (`float`) – 一阶最优的终止容忍度（默认：1e-5）
- tolerance_change (`float`) – 在函数值/参数变化量上的终止容忍度（默认：1e-9）
- history_size (`int`) – 更新历史的大小（默认：100）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False) [source]

实现RMSprop算法。

由G. H `int` on在他的课程中提出.

中心版本首次出现在Generating Sequences With Recurrent Neural Networks.

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float` , 可选) – 学习率（默认：1e-2）
- momentum (`float` , 可选) – 动量因子（默认：0）
- alpha (`float` , 可选) – 平滑常数（默认：0.99）
- eps (`float` , 可选) – 为了增加数值计算的稳定性而加到分母里的项（默认：1e-8）
- centered (`bool` , 可选) – 如果为True，计算中心化的RMSProp，并且用它的方差预测值对梯度进行归一化
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认: 0）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.Rprop(params, lr=0.01, etas=(0.5, 1.2), step_sizes=(1e-06, 50))[source]

实现弹性反向传播算法。

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float` , 可选) – 学习率（默认：1e-2）
- etas (Tuple[`float` , `float`], 可选) – 一对（etaminus，etaplis）, 它们分别是乘法的增加和减小的因子（默认：0.5，1.2）
- step_sizes (Tuple[`float` , `float`], 可选) – 允许的一对最小和最大的步长（默认：1e-6，50）

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

class torch.optim.SGD(params, lr=, momentum=0, dampening=0, weight_decay=0, nesterov=False)[source]

实现随机梯度下降算法（momentum可选）。

Nesterov动量基于On the importance of initialization and momentum in deep learning中的公式.

参数：

- params (iterable) – 待优化参数的iterable或者是定义了参数组的dict
- lr (`float`) – 学习率
- momentum (`float` , 可选) – 动量因子（默认：0）
- weight_decay (`float` , 可选) – 权重衰减（L2惩罚）（默认：0）
- dampening (`float` , 可选) – 动量的抑制因子（默认：0）
- nesterov (`bool` , 可选) – 使用Nesterov动量（默认：False）

例子：

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Note

带有动量/Nesterov的SGD的实现稍微不同于Sutskever等人以及其他框架中的实现。

考虑动量的具体情况，更新可以写成

$v=\rho*v+g$

$p=p-lr*v$

其中，p、g、v和ρ分别是参数、梯度、速度和动量。

这跟Sutskever等人以及其他框架的实现是相反的，它们采用这样的更新

$v=\rho*v+lr*g$

$p=p-v$

Nesterov的版本也类似地被修改了。

step(closure) [source]

进行单次优化 (参数更新).

参数：

- closure (`callable`) – 一个重新评价模型并返回loss的闭包，对于大多数参数来说是可选的。

torch.nn.init

```
torch.nn.init.calculate_gain(nonlinearity,param=None)
```

对于给定的非线性函数，返回推荐的增益值。这些值如下所示：

nonlinearity	gain
linear	1
conv{1,2,3}d	1
sigmoid	1
tanh	5/3
relu	sqrt(2)
leaky_relu	sqrt(2/(1+negative_slope^2))

参数：

- **nonlinearity** - 非线性函数（`nn.functional`名称）
- **param** - 非线性函数的可选参数

例子：

```
>>> gain = nn.init.gain('leaky_relu')
```

```
torch.nn.init.uniform(tensor, a=0, b=1)
```

从均匀分布U(a, b)中生成值，填充输入的张量或变量

参数：

- **tensor** - n维的torch.Tensor
- **a** - 均匀分布的下界
- **b** - 均匀分布的上界

例子

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.uniform(w)
```

```
torch.nn.init.normal(tensor, mean=0, std=1)
```

从给定均值和标准差的正态分布N(mean, std)中生成值，填充输入的张量或变量

参数：

- **tensor** – n维的torch.Tensor
- **mean** – 正态分布的均值
- **std** – 正态分布的标准差

例子

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.normal(w)
```

```
torch.nn.init.constant(tensor, val)
```

用*val*的值填充输入的张量或变量

参数：

- **tensor** – n维的torch.Tensor或autograd.Variable
- **val** – 用来填充张量的值

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.constant(w)
```

```
torch.nn.init.eye(tensor)
```

用单位矩阵来填充2维输入张量或变量。在线性层尽可能多的保存输入特性。

参数：

- **tensor** – 2维的torch.Tensor或autograd.Variable

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.eye(w)
```

```
torch.nn.init.dirac(tensor)
```

用Dirac δ 函数来填充{3, 4, 5}维输入张量或变量。在卷积层尽可能多的保存输入通道特性。

参数：

- **tensor** – {3, 4, 5}维的torch.Tensor或autograd.Variable

例子：

```
>>> w = torch.Tensor(3, 16, 5, 5)
>>> nn.init.dirac(w)
```

```
torch.nn.init.xavier_uniform(tensor, gain=1)
```

根据Glorot, X.和Bengio, Y.在“Understanding the difficulty of training deep feedforward neural networks”中描述的方法，用一个均匀分布生成值，填充输入的张量或变量。结果张量中的值采样自U(-a, a)，其中a= gain *sqrt(2/(fan_in + fan_out))* sqrt(3). 该方法也被称为Glorot initialisation

参数：

- **tensor** – n维的torch.Tensor
- **gain** - 可选的缩放因子

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.xavier_uniform(w, gain=math.sqrt(2.0))
```

```
torch.nn.init.xavier_normal(tensor, gain=1)
```

根据Glorot, X.和Bengio, Y. 于2010年在“Understanding the difficulty of training deep feedforward neural networks”中描述的方法，用一个正态分布生成值，填充输入的张量或变量。结果张量中的值采样自均值为0，标准差为gain * sqrt(2/(fan_in + fan_out))的正态分布。也被称为Glorot initialisation.

参数：

- **tensor** – n维的torch.Tensor
- **gain** - 可选的缩放因子

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.xavier_normal(w)
```

```
torch.nn.init.kaiming_uniform(tensor, a=0, mode='fan_in')
```

根据He, K等人于2015年在“Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification”中描述的方法，用一个均匀分布生成值，填充输入的张量或变量。结果张量中的值采样自U(-bound, bound)，其中bound = sqrt(2/((1 + a^2) *fan_in))* sqrt(3)。也被称为He initialisation.

参数：

- **tensor** – n维的torch.Tensor或autograd.Variable
- **a** -这层之后使用的rectifier的斜率系数（ReLU的默认值为0）
- **mode** -可以为“fan_in”（默认）或“fan_out”。“fan_in”保留前向传播时权值方差的量级，“fan_out”保留反向传播时的量级。

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.kaiming_uniform(w, mode='fan_in')
```

```
torch.nn.init.kaiming_normal(tensor, a=0, mode='fan_in')
```

根据He, K等人在“Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification”中描述的方法，用一个正态分布生成值，填充输入的张量或变量。结果张量中的值采样自均值为0，标准差为sqrt(2/((1 + a^2) * fan_in))的正态分布。

参数：

- **tensor** – n维的torch.Tensor或 autograd.Variable
- **a** -这层之后使用的rectifier的斜率系数（ReLU的默认值为0）
- **mode** -可以为“fan_in”（默认）或“fan_out”。“fan_in”保留前向传播时权值方差的量级，“fan_out”保留反向传播时的量级。

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.kaiming_normal(w, mode='fan_out')
```

```
torch.nn.init.orthogonal(tensor, gain=1)
```

用（半）正交矩阵填充输入的张量或变量。输入张量必须至少是2维的，对于更高维度的张量，超出的维度会被展平，视作行等于第一个维度，列等于稀疏矩阵乘积的2维表示。其中非零元素生成自均值为0，标准差为std的正态分布。

参考：Saxe, A等人(2013)的“Exact solutions to the nonlinear dynamics of learning in deep linear neural networks”

参数：

- **tensor** – n维的torch.Tensor或 autograd.Variable，其中n>=2
- **gain** -可选

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.orthogonal(w)
```

```
torch.nn.init.sparse(tensor, sparsity, std=0.01)
```

将2维的输入张量或变量当做稀疏矩阵填充，其中非零元素根据一个均值为0，标准差为std的正态分布生成。 参考Martens, J.(2010)的 “Deep learning via Hessian-free optimization”.

参数：

- **tensor** – n维的torch.Tensor或autograd.Variable
- **sparsity** - 每列中需要被设置成零的元素比例
- **std** - 用于生成非零值的正态分布的标准差

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.sparse(w, sparsity=0.1)
```

torch.multiprocessing

封装了 `multiprocessing` 模块。用于在相同数据的不同进程中共享视图。

一旦张量或者存储被移动到共享单元(见 `share_memory_()`),它可以不需要任何其他复制操作的发送到其他的进程中。

这个API与原始模型完全兼容，为了让张量通过队列或者其他机制共享，移动到内存中，我们可以

由原来的 `import multiprocessing` 改为 `import torch.multiprocessing` 。

由于API的相似性，我们没有记录这个软件包的大部分内容，我们建议您参考原始模块的非常好的文档。

warning： 如果主要的进程突然退出(例如，因为输入信号)，Python中的 `multiprocessing` 有时会不能清理他的子节点。

这是一个已知的警告，所以如果您在中断解释器后看到任何资源泄漏，这可能意味着这刚刚发生在您身上。

Strategy management

```
torch.multiprocessing.get_all_sharing_strategies()
```

返回一组由当前系统所支持的共享策略

```
torch.multiprocessing.get_sharing_strategy()
```

返回当前策略共享CPU中的张量。

```
torch.multiprocessing.set_sharing_strategy(new_strategy)
```

设置共享CPU张量的策略

参数: new_strategy(str)-被选中策略的名字。应当是 `get_all_sharing_strategies()` 中值当中的一个。

Sharing CUDA tensors

共享CUDA张量进程只支持Python3，使用 `spawn` 或者 `forkserver` 开始方法。

Python2中的`multiprocessing`只能使用`fork`创建子进程，并且不被CUDA支持。

`warning：` CUDA API要求导出到其他进程的分配一直保持有效，只要它们被使用。

你应该小心，确保您共享的CUDA张量不要超出范围。

这不应该是共享模型参数的问题，但传递其他类型的数据应该小心。请注意，此限制不适用于共享CPU内存。

Sharing strategies

本节简要概述了不同的共享策略如何工作。

请注意，它仅适用于CPU张量 - CUDA张量将始终使用CUDA API，因为它们是唯一的共享方式。

File descriptor-`file_descripor`

`NOTE：` 这是默认策略（除了不支持的MacOS和OS X）。

此策略将使用文件描述符作为共享内存句柄。当存储被移动到共享内存中，一个由`shm_open`获得的文件描述符被缓存，

并且当它将被发送到其他进程时，文件描述符将被传送（例如通过UNIX套接字）。

接收者也将缓存文件描述符，并且`mmap`它，以获得对存储数据的共享视图。

请注意，如果要共享很多张量，则此策略将保留大量文件描述符。

如果你的系统对打开的文件描述符数量有限制，并且无法提高，你应该使用`file_system`策略。

File system -file_system

这个策略将提供文件名称给`shm_open`去定义共享内存区域。

该策略不需要缓存从其获得的文件描述符的优点，但是容易发生共享内存泄漏。

该文件创建后不能被删除，因为其他进程需要访问它以打开其视图。

如果进程崩溃或死机，并且不能调用存储析构函数，则文件将保留在系统中。

这是非常严重的，因为它们在系统重新启动之前不断使用内存，或者手动释放它们。

为了记录共享内存文件泄露数量，`torch.multiprocessing`将产生一个守护进程叫做`torch_shm_manager`

将自己与当前进程组隔离，并且将跟踪所有共享内存分配。一旦连接到它的所有进程退出，

它将等待一会儿，以确保不会有新的连接，并且将遍历该组分配的所有共享内存文件。

如果发现它们中的任何一个仍然存在，它们将被释放。我们已经测试了这种方法，并且它已被证明对于各种故障都是稳健的。

如果你的系统有足够高的限制，并且 `file_descriptor` 是被支持的策略，我们不建议切换到这个。

遗产包 - torch.legacy

此包中包含从Lua Torch移植来的代码。

为了可以使用现有的模型并且方便当前Lua Torch使用者过渡，我们创建了这个包。 可以在`torch.legacy.nn`中找到`nn`代码，并在`torch.legacy.optim`中找到`optim`代码。 API应该完全匹配Lua Torch。

torch.cuda

该包增加了对CUDA张量类型的支持，实现了与CPU张量相同的功能，但使用GPU进行计算。

它是懒惰的初始化，所以你可以随时导入它，并使用 `is_available()` 来确定系统是否支持CUDA。

CUDA语义中有关于使用CUDA的更多细节。

```
torch.cuda.current_blas_handle()
```

返回cublasHandle_t指针，指向当前cuBLAS句柄

```
torch.cuda.current_device()
```

返回当前所选设备的索引。

```
torch.cuda.current_stream()
```

返回一个当前所选的 `Stream`

```
class torch.cuda.device(idx)
```

上下文管理器，可以更改所选设备。

参数：

- **idx** (*int*) – 设备索引选择。如果这个参数是负的，则是无效操作。

```
torch.cuda.device_count()
```

返回可得到的GPU数量。

```
class torch.cuda.device_of(obj)
```

将当前设备更改为给定对象的上下文管理器。

可以使用张量和存储作为参数。如果给定的对象不是在GPU上分配的，这是一个无效操作。

参数：

- **obj** (*Tensor* or *Storage*) – 在选定设备上分配的对象。

```
torch.cuda.is_available()
```

返回一个bool值，指示CUDA当前是否可用。

```
torch.cuda.set_device(device)
```

设置当前设备。

不鼓励使用此函数来设置。在大多数情况下，最好使用 `CUDA_VISIBLE_DEVICES` 环境变量。

参数：

- **device** (*int*) – 所选设备。如果此参数为负，则此函数是无效操作。

```
torch.cuda.stream(stream)
```

选择给定流的上下文管理器。

在其上下文中排队的所有CUDA核心将在所选流上入队。

参数：

- **stream** (*Stream*) – 所选流。如果是 `None` ，则这个管理器是无效的。

```
torch.cuda.synchronize()
```

等待当前设备上所有流中的所有核心完成。

交流集

```
torch.cuda.comm.broadcast(tensor, devices)
```

向一些GPU广播张量。

参数：

- **tensor** (*Tensor*) – 将要广播的张量
- **devices** (*Iterable*) – 一个可以广播的设备的迭代。注意，它的形式应该像（src，dst1，dst2，...），其第一个元素是广播来源的设备。

返回： 一个包含张量副本的元组，放置在与设备的索引相对应的设备上。

```
torch.cuda.comm.reduce_add(inputs, destination=None)
```

将来自多个GPU的张量相加。

所有输入应具有匹配的形状。

参数：

- **inputs** (*Iterable[Tensor]*) – 要相加张量的迭代
- **destination** (*int*, optional) – 将放置输出的设备（默认值：当前设备）。

返回： 一个包含放置在 `destination` 设备上的所有输入的元素总和的张量。

```
torch.cuda.comm.scatter(tensor, devices, chunk_sizes=None, dim=0, streams=None)
```

打散横跨多个GPU的张量。

参数：

- **tensor** (*Tensor*) – 要分散的张量
- **devices** (*Iterable[int]*) – int的迭代，指定哪些设备应该分散张量。
- **chunk_sizes** (*Iterable[int]*, optional) – 要放置在每个设备上的块大小。它应该匹配 `devices` 的长度并且总和为 `tensor.size(dim)` 。 如果没有指定，张量将被分成相等的块。
- **dim** (*int*, optional) – 沿着这个维度来chunk张量

返回： 包含 `tensor` 块的元组，分布在给定的 `devices` 上。

```
torch.cuda.comm.gather(tensors, dim=0, destination=None)
```

从多个GPU收集张量。

张量尺寸在不同于 `dim` 的所有维度上都应该匹配。

参数：

- **tensors** (*Iterable[Tensor]*) – 要收集的张量的迭代。
- **dim** (*int*) – 沿着此维度张量将被连接。
- **destination** (*int*, optional) – 输出设备（-1表示CPU，默认值：当前设备）。

返回： 一个张量位于 `destination` 设备上，这是沿着 `dim` 连接 `tensors` 的结果。

流和事件

```
class torch.cuda.Stream
```

CUDA流的包装。

参数：

- **device** (*int*, optional) – 分配流的设备。
- **priority** (*int*, optional) – 流的优先级。较低的数字代表较高的优先级。

> query()

检查所有提交的工作是否已经完成。

返回： 一个布尔值，表示此流中的所有核心是否完成。

> record_event(event=None)

记录一个事件。

参数： **event** (*Event*, optional) – 要记录的事件。如果没有给出，将分配一个新的。 返回： 记录的事件。

> synchronize()

等待此流中的所有核心完成。

> wait_event(event)

将所有未来的工作提交到流等待事件。

参数： **event** (*Event*) – 等待的事件

> wait_stream(stream)

与另一个流同步。

提交到此流的所有未来工作将等待直到所有核心在调用完成时提交给给定的流。

```
class torch.cuda.Event(enable_timing=False, blocking=False, interprocess=False, _handl
e=None)
```

CUDA事件的包装。

参数：

- **enable_timing** (*bool*) – 指示事件是否应该测量时间（默认值：False）
- **blocking** (*bool*) – 如果为true， `wait()` 将被阻塞（默认值：False）
- **interprocess** (*bool*) – 如果为true，则可以在进程之间共享事件（默认值：False）

> elapsed_time(end_event)

返回事件记录之前经过的时间。

> ipc_handle()

返回此事件的IPC句柄。

> query()

检查事件是否已被记录。

返回： 一个布尔值，指示事件是否已被记录。

> record(stream=None)

记录给定流的事件。

> synchronize()

与事件同步。

> wait(stream=None)

使给定的流等待事件。

torch.utils.ffi

```
torch.utils.ffi.create_extension(name, headers, sources, verbose=True, with_cuda=False
, package=False, relative_to='.', **kwargs)
```

创建并配置一个cffi.FFI对象,用于PyTorch的扩展。

参数：

- **name** (*str*) – 包名。可以是嵌套模块，例如 `.ext.my_lib` 。
- **headers** (*str* or List[*str*]) – 只包含导出函数的头文件列表
- **sources** (List[*str*]) – 用于编译的sources列表
- **verbose** (*bool*, optional) – 如果设置为False，则不会打印输出（默认值： `True` ）。
- **with_cuda** (*bool*, optional) – 设置为True以使用CUDA头文件进行编译（默认值： `False` ）。
- **package** (*bool*, optional) – 设置为True以在程序包模式下构建（对于要作为pip程序包安装的模块）（默认值： `False` ）。
- **relative_to** (*str*, optional) –构建文件的路径。 `package` 为 `True` 时需要。最好使用 `__file__` 作为参数。
- **kwargs** – 传递给ffi以声明扩展的附加参数。有关详细信息，请参阅Extension API reference。

torch.utils.data

```
class torch.utils.data.Dataset
```

表示Dataset的抽象类。

所有其他数据集都应该进行子类化。所有子类应该override `__len__` 和 `__getitem__` ，前者提供了数据集的大小，后者支持整数索引，范围从0到len(self)。

```
class torch.utils.data.TensorDataset(data_tensor, target_tensor)
```

包装数据和目标张量的数据集。

通过沿着第一个维度索引两个张量来恢复每个样本。

参数：

- **data_tensor** (*Tensor*) － 包含样本数据
- **target_tensor** (*Tensor*) － 包含样本目标（标签）

```
class torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
num_workers=0, collate_fn=<function default_collate>, pin_memory=False, drop_last=Fals
e)
```

数据加载器。组合数据集和采样器，并在数据集上提供单进程或多进程迭代器。

参数：

- **dataset** (*Dataset*) – 加载数据的数据集。
- **batch_size** (*int*, optional) – 每个batch加载多少个样本(默认: 1)。
- **shuffle** (*bool*, optional) – 设置为 `True` 时会在每个epoch重新打乱数据(默认: False).
- **sampler** (*Sampler*, optional) – 定义从数据集中提取样本的策略。如果指定，则忽略 `shuffle` 参数。
- **num_workers** (*int*, optional) – 用多少个子进程加载数据。0表示数据将在主进程中加载(默认: 0)
- **collate_fn** (*callable*, optional) –
- **pin_memory** (*bool*, optional) –
- **drop_last** (*bool*, optional) – 如果数据集大小不能被batch size整除，则设置为True后可删除最后一个不完整的batch。如果设为False并且数据集的大小不能被batch size整除，则最后一个batch将更小。(默认: False)

```
class torch.utils.data.sampler.Sampler(data_source)
```

所有采样器的基础类。

每个采样器子类必须提供一个 `__iter__` 方法，提供一种迭代数据集元素的索引的方法，以及返回迭代器长度的 `__len__` 方法。

```
class torch.utils.data.sampler.SequentialSampler(data_source)
```

样本元素顺序排列，始终以相同的顺序。

参数：

- **data_source** (*Dataset*) – 采样的数据集。

```
class torch.utils.data.sampler.RandomSampler(data_source)
```

样本元素随机，没有替换。

参数：

- **data_source** (*Dataset*) – 采样的数据集。

```
class torch.utils.data.sampler.SubsetRandomSampler(indices)
```

样本元素从指定的索引列表中随机抽取，没有替换。

参数：

- **indices** (*list*) – 索引的列表

```
class torch.utils.data.sampler.WeightedRandomSampler(weights, num_samples, replacement
=True)
```

样本元素来自于[0,..,len(weights)-1]，给定概率（weights）。

参数：

- **weights** (*list*) – 权重列表。没必要加起来为1
- **num_samples** (*int*) – 抽样数量

torch.utils.model_zoo

```
torch.utils.model_zoo.load_url(url, model_dir=None)
```

在给定URL上加载Torch序列化对象。

如果对象已经存在于 *model_dir* 中，则将被反序列化并返回。URL的文件名部分应遵循命名约定 `filename-<sha256>.ext` ，其中 `<sha256>` 是文件内容的SHA256哈希的前八位或更多位数字。哈希用于确保唯一的名称并验证文件的内容。

model_dir 的默认值为 `$TORCH_HOME/models` ，其中 `$TORCH_HOME` 默认为 `~/.torch` 。可以使用 `$TORCH_MODEL_ZOO` 环境变量来覆盖默认目录。

参数：

- **url** (*string*) - 要下载对象的URL
- **model_dir** (*string*, optional) - 保存对象的目录

例子：

```
>>> state_dict = torch.utils.model_zoo.load_url('https://s3.amazonaws.com/pytorch/models/resnet18-5c106cde.pth')
```

torchvision

`torchvision` 包 包含了目前流行的数据集，模型结构和常用的图片转换工具。

torchvision.datasets

`torchvision.datasets` 中包含了以下数据集

- MNIST
- COCO（用于图像标注和目标检测）(Captioning and Detection)
- LSUN Classification
- ImageFolder
- Imagenet-12
- CIFAR10 and CIFAR100
- STL10

`Datasets` 拥有以下 `API`：

`__getitem__` `__len__`

由于以上 `Datasets` 都是 `torch.utils.data.Dataset` 的子类，所以，他们也可以通过 `torch.utils.data.DataLoader` 使用多线程（python的多进程）。

举例说明：
`torch.utils.data.DataLoader(coco_cap, batch_size=args.batchSize, shuffle=True, num_workers=`

在构造函数中，不同的数据集直接的构造函数会有些许不同，但是他们共同拥有 `keyword` 参数。 In the constructor, each dataset has a slightly different API as needed, but they all take the keyword args:

- `transform` ： 一个函数，原始图片作为输入，返回一个转换后的图片。（详情请看下面关于 `torchvision-tranform` 的部分）

- `target_transform` - 一个函数，输入为 `target` ，输出对其的转换。例子，输入的是图片标注的 `string` ，输出为 `word` 的索引。

MNIST

```
dset.MNIST(root, train=True, transform=None, target_transform=None, download=False
)
```

参数说明：

- root : `processed/training.pt` 和 `processed/test.pt` 的主目录
- train : `True` = 训练集, `False` = 测试集
- download : `True` = 从互联网上下载数据集，并把数据集放在 `root` 目录下. 如果数据集之前下载过，将处理过的数据（minist.py中有相关函数）放在 `processed` 文件夹下。

COCO

需要安装COCO API

图像标注:

```
dset.CocoCaptions(root="dir where images are", annFile="json annotation file", [transf
orm, target_transform])
```

例子:

```
import torchvision.datasets as dset
import torchvision.transforms as transforms
cap = dset.CocoCaptions(root = 'dir where images are',
                        annFile = 'json annotation file',
                        transform=transforms.ToTensor())

print('Number of samples: ', len(cap))
img, target = cap[3] # load 4th sample

print("Image Size: ", img.size())
print(target)
```

输出:

```
Number of samples: 82783
Image Size: (3L, 427L, 640L)
[u'A plane emitting smoke stream flying over a mountain.',
u'A plane darts across a bright blue sky behind a mountain covered in snow',
u'A plane leaves a contrail above the snowy mountain top.',
u'A mountain that has a plane flying overheard in the distance.',
u'A mountain view with a plume of smoke in the background']
```

检测:

```
dset.CocoDetection(root="dir where images are", annFile="json annotation file", [trans
form, target_transform])
```

LSUN

```
dset.LSUN(db_path, classes='train', [transform, target_transform])
```

参数说明：

- db_path = 数据集文件的根目录
- classes = ‘train’ (所有类别, 训练集), ‘val’ (所有类别, 验证集), ‘test’ (所有类别, 测试集) [‘bedroom_train’, ‘church_train’, …] : a list of categories to load

ImageFolder

一个通用的数据加载器，数据集中的数据以以下方式组织 ``` root/dog/xxx.png root/dog/xxy.png root/dog/xxz.png

root/cat/123.png root/cat/nsdf3.png root/cat/asd932_.png

```
```python
dset.ImageFolder(root="root folder path", [transform, target_transform])
```

他有以下成员变量:

- self.classes - 用一个list保存 类名
- self.class_to_idx - 类名对应的 索引
- self.imgs - 保存(img-path, class) tuple的list

Imagenet-12

This is simply implemented with an ImageFolder dataset.

The data is preprocessed as described here

Here is an example

CIFAR

```
dset.CIFAR10(root, train=True, transform=None, target_transform=None, download=False)

dset.CIFAR100(root, train=True, transform=None, target_transform=None, download=False)
```

参数说明：

- root : `cifar-10-batches-py` 的根目录
- train : `True` = 训练集, `False` = 测试集
- download : `True` = 从互联上下载数据，并将其放在`root`目录下。如果数据集已经下载，什么都不干。

STL10

```
dset.STL10(root, split='train', transform=None, target_transform=None, download=False)
```
```

参数说明：

- root : `stl10_binary`的根目录
- split : 'train' = 训练集, 'test' = 测试集, 'unlabeled' = 无标签数据集, 'train+unlabeled' = 训练 + 无标签数据集 (没有标签的标记为-1)
- download : `True` = 从互联上下载数据，并将其放在`root`目录下。如果数据集已经下载，什么都不干。

torchvision.models

`torchvision.models` 模块的 子模块中包含以下模型结构。

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet You can construct a model with random weights by calling its constructor:

你可以使用随机初始化的权重来创建这些模型。

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
squeezenet = models.squeezenet1_0()
densenet = models.densenet_161()
```

We provide pre-trained models for the ResNet variants and AlexNet, using the PyTorch torch.utils.model_zoo. These can constructed by passing pretrained=True: 对于 `ResNet variants` 和 `AlexNet` ，我们也提供了预训练(`pre-trained`)的模型。

```
import torchvision.models as models
#pretrained=True就可以使用预训练的模型
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
```

ImageNet 1-crop error rates (224x224)

Network	Top-1 error	Top-5 error
ResNet-18	30.24	10.92
ResNet-34	26.70	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
Inception v3	22.55	6.44
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20

torchvision.models.alexnet(pretrained=False, ** kwargs)

`AlexNet` 模型结构 paper地址

- pretrained (bool) – `True` , 返回在ImageNet上训练好的模型。

torchvision.models.resnet18(pretrained=False, ** kwargs)

构建一个`resnet18`模型

- pretrained (bool) – `True` , 返回在ImageNet上训练好的模型。

torchvision.models.resnet34(pretrained=False, ** kwargs)

构建一个`ResNet-34` 模型.

Parameters: pretrained (bool) – `True`, 返回在ImageNet上训练好的模型。

torchvision.models.resnet50(pretrained=False, ** kwargs)

构建一个`ResNet-50`模型

- pretrained (bool) – `True`, 返回在ImageNet上训练好的模型。

torchvision.models.resnet101(pretrained=False, ** kwargs)

Constructs a ResNet-101 model.

- pretrained (bool) – `True`, 返回在ImageNet上训练好的模型。

torchvision.models.resnet152(pretrained=False, ** kwargs)

Constructs a ResNet-152 model.

- pretrained (bool) – `True`, 返回在ImageNet上训练好的模型。

torchvision.models.vgg11(pretrained=False, ** kwargs)

VGG 11-layer model (configuration “A”)

- pretrained (bool) – `True`, 返回在ImageNet上训练好的模型。

torchvision.models.vgg11_bn(** kwargs)

VGG 11-layer model (configuration “A”) with batch normalization

torchvision.models.vgg13(pretrained=False, ** kwargs)

VGG 13-layer model (configuration “B”)

- pretrained (bool) – `True` , 返回在ImageNet上训练好的模型。

torchvision.models.vgg13_bn(** kwargs)

VGG 13-layer model (configuration “B”) with batch normalization

torchvision.models.vgg16(pretrained=False, ** kwargs)

VGG 16-layer model (configuration “D”)

Parameters: pretrained (bool) – If True, returns a model pre-trained on ImageNet

torchvision.models.vgg16_bn(** kwargs)

VGG 16-layer model (configuration “D”) with batch normalization

torchvision.models.vgg19(pretrained=False, ** kwargs)

VGG 19-layer model (configuration “E”)

- pretrained (bool) – `True` , 返回在ImageNet上训练好的模型。

torchvision.models.vgg19_bn(** kwargs)

VGG 19-layer model (configuration ‘E’) with batch normalization

pytorch torchvision transform

对PIL.Image进行变换

class torchvision.transforms.Compose(transforms)

将多个`transform`组合起来使用。

`transforms`：由`transform`构成的列表. 例子：

```
transforms.Compose([
     transforms.CenterCrop(10),
     transforms.ToTensor(),
 ])
```

class torchvision.transforms.Scale(size, interpolation=2)

将输入的`PIL.Image`重新改变大小成给定的`size`，`size`是最小边的边长。举个例子，如果原图的`height>width`,那么改变大小后的图片大小是`(size*height/width, size)`。用例:

```
from torchvision import transforms
from PIL import Image
crop = transforms.Scale(12)
img = Image.open('test.jpg')

print(type(img))
print(img.size)

croped_img=crop(img)
print(type(croped_img))
print(croped_img.size)
```

```
<class 'PIL.PngImagePlugin.PngImageFile'>
(10, 10)
<class 'PIL.Image.Image'>
(12, 12)
```

class torchvision.transforms.CenterCrop(size)

将给定的`PIL.Image`进行中心切割，得到给定的`size`，`size`可以是`tuple`，`(target_height, target_width)`。`size`也可以是一个`Integer`，在这种情况下，切出来的图片的形状是正方形。

class torchvision.transforms.RandomCrop(size, padding=0)

切割中心点的位置随机选取。 `size` 可以是 `tuple` 也可以是 `Integer` 。

class torchvision.transforms.RandomHorizontalFlip

随机水平翻转给定的 `PIL.Image` ,概率为 `0.5` 。即：一半的概率翻转，一半的概率不翻转。

class torchvision.transforms.RandomSizedCrop(size, interpolation=2)

先将给定的 `PIL.Image` 随机切，然后再 `resize` 成给定的 `size` 大小。

class torchvision.transforms.Pad(padding, fill=0)

将给定的 `PIL.Image` 的所有边用给定的 `pad value` 填充。 `padding：` 要填充多少像素 `fill：` 用什么值填充 例子：

```
from torchvision import transforms
from PIL import Image
padding_img = transforms.Pad(padding=10, fill=0)
img = Image.open('test.jpg')

print(type(img))
print(img.size)

padded_img=padding(img)
print(type(padded_img))
print(padded_img.size)
```

```
<class 'PIL.PngImagePlugin.PngImageFile'>
(10, 10)
<class 'PIL.Image.Image'>
(30, 30) #由于上下左右都要填充10个像素，所以填充后的size是(30,30)
```

对Tensor进行变换

class torchvision.transforms.Normalize(mean, std)

给定均值： `(R,G,B)` 方差： `（R，G，B）` ，将会把 `Tensor` 正则化。
即： `Normalized_image=(image-mean)/std` 。

Conversion Transforms

class torchvision.transforms.ToTensor

把一个取值范围是 `[0,255]` 的 `PIL.Image` 或者 `shape` 为 `(H,W,C)` 的 `numpy.ndarray` ，转换成形状为 `[C,H,W]` ，取值范围是 `[0,1.0]` 的 `torch.FloadTensor`

```
data = np.random.randint(0, 255, size=300)
img = data.reshape(10,10,3)
print(img.shape)
img_tensor = transforms.ToTensor()(img) # 转换成tensor
print(img_tensor)
```

class torchvision.transforms.ToPILImage

将 `shape` 为 `(C,H,W)` 的 `Tensor` 或 `shape` 为 `(H,W,C)` 的 `numpy.ndarray` 转换成 `PIL.Image` ，值不变。

通用变换

class torchvision.transforms.Lambda(lambd)

使用 `lambd` 作为转换器。

torchvision.utils

torchvision.utils.make_grid(tensor, nrow=8, padding=2, normalize=False, range=None, scale_each=False)

猜测，用来做 `雪碧图的` （ `sprite image` ）。

给定 `4D mini-batch Tensor` ， 形状为 `(B x C x H x W)`,或者一个`a list of image` ，做成一个`size`为`(B / nrow, nrow)`的雪碧图。

- normalize=True ，会将图片的像素值归一化处理
- 如果 range=(min, max)， min和max是数字，那么`min` ，`max`用来规范化`image`
- scale_each=True ，每个图片独立规范化，而不是根据所有图片的像素最大最小值来规范化

Example usage is given in this notebook

torchvision.utils.save_image(tensor, filename, nrow=8, padding=2, normalize=False, range=None, scale_each=False)

将给定的`Tensor`保存成image文件。如果给定的是`mini-batch tensor` ，那就用`make-grid`做成雪碧图，再保存。

致谢

本项目贡献者如下：

文档翻译

贡献者	页面	章节
ycszen	主页	
ycszen	说明	自动求导机制
ycszen	说明	CUDA语义
KeithYin	说明	扩展PyTorch
ycszen	说明	多进程最佳实践
ycszen	说明	序列化语义
koshinryuu	package参考	torch
weigp	package参考	torch.Tensor
kophy	package参考	torch.Storage
KeithYin	package参考	torch.nn/Parameters
KeithYin	package参考	torch.nn/Containers
yichuan9527	package参考	torch.nn/Convolution Layers
yichuan9527	package参考	torch.nn/Pooling Layers
swordspoet	package参考	torch.nn/Non-linear Activations
XavierLin	package参考	torch.nn/Normalization layers
KeithYin	package参考	torch.nn/Recurrent layers
	package参考	torch.nn/Linear layers
	package参考	torch.nn/Dropout layers
	package参考	torch.nn/Distance functions
KeithYin	package参考	torch.nn/Loss functions
KeithYin	package参考	torch.nn/Vision layers
KeithYin	package参考	torch.nn/Multi-GPU layers
KeithYin	package参考	torch.nn/Utilities
ycszen	package参考	torch.nn.functional/Convolution functions

ycszen	package参考	torch.nn.functional/Pooling function
ycszen	package参考	torch.nn.functional/Non-linear activations functions
ycszen	package参考	torch.nn.functional/Normalization functions
dyl745001196	package参考	torch.nn.functional/Linear functions
dyl745001196	package参考	torch.nn.functional/Dropout functions
dyl745001196	package参考	torch.nn.functional/Distance functions
tfygg	package参考	torch.nn.functinal/Loss functions
KeithYin	package参考	torch.nn.functional/Vision functions
kophy	package参考	torch.nn.init
KeithYin	package参考	torch.autograd
songbo.han	package参考	torch.multiprocessing
ZijunDeng	package参考	torch.optim
ycszen	pacakge参考	torch.legacy
ycszen	package参考	torch.cuda
ycszen	pacakge参考	torch.utils.ffi
ycszen	package参考	torch.utils.model_zoo
ycszen	package参考	torch.utils.data
KeithYin	torchvision参考	torchvision
KeithYin	torchvision参考	torchvision.datasets
KeithYin	torchvision参考	torchvision.models
KeithYin	torchvision参考	torchvision.transforms
KeithYin	torchvision参考	torchvision.utils
ycszen	致谢	